FrankHB talk C/C++

Mingkai Zhou

Published with GitBook



目錄

Introduction and Copyright 0
术语和文献列表 1
ISO C/C++ 基础总结 2
概念和纠错 3
《高质量C++/C编程指南》陷阱 3.1
字符串和字符串的长度 3.2
面向对象和所谓的“面向过程” 3.3
关于 main 函数的原型和返回值 3.4
C11 & C++11 的赋值相关表达式求值 3.5
变量、全局变量及其它 3.6
早期论述： 3.7
建议慎用“变量”一词 3.7.1
C语言的范畴里根本就无所谓“全局”“变量”？ 3.7.2
关于异常处理的一些话题 3.8
《高质量C++/C编程指南》陷阱 2 3.9
何勤《轻松学习C程序设计》简评 3.10
什么叫语法(syntax) 3.11
特性和设计 4
C & C++广义类型系统缺陷 4.1
转移 vs 复制 4.2
正确地黑 C 4.3
C++ 设计缺陷 4.4
为什么指针是个糟糕的语言特性 4.5
早期论述： 4.6
[1] 4.6.1
[2] 4.6.2
[3] 4.6.3
语用和科普 5
[FAQ]MinGW vs MinGW-W64及其它 5.1

帝球的**C/C++**文档

This is a book about some programing languages and something about their theory.

Copyright

术语和文献列表

本文用于对相关文章内容的正式引用提供解释性补充参照。

C 语言标准

ANSI C89 指 ANSI X3.159-1989 ，后来被采纳为 ISO/IEC 9899:1990 ，通称 C90 。两者正文仅有格式变化（另外 C90 不包含 Rationale ）。

C99 或 ISO C99 指 ISO/IEC 9899:1999 ，被 ANSI 于 2000 年 5 月采纳。

C11 或 ISO C11 指 ISO/IEC 9899:2011 ，是第三版正式C语言国际语言标准，也被称为 C1X 。

ANSI C 一般即 ANSI C89 。实际上也只有这版标准是先 ANSI 后 ISO 。

C++ 标准

C++98 或 ISO C++98 指 ISO/IEC 14882:1998 ，是第一个 C++ 语言国际标准。

C++03 或 ISO C++03 指 ISO/IEC 14882:2003 ，是第二版 C++ 语言国际标准，对上一版只有小的修正。

C++11 或 ISO C++11 指 ISO/IEC 14882:2011 ，是第三版 C++ 语言国际标准，有较大改动。在出版之前，曾经有较长时间的延期，被称为 C++0x 。

C++14 或 ISO C++14 指 ISO/IEC 14882:2014 ，是第四版 C++ 语言国际标准，有一定修正，之前称为 C++1y 。

C++17 或 ISO C++17 指预定出版的 ISO/IEC 14882:2017 ，是 C++ 语言国际标准的一个主要版本，被称为 C++1z 。

ISO C/C++ 基础总结

本文档意为避免 C 和 C++ 语言中的一些陷阱的通用指引及撰写相关 FAQ 的权威参考的要点注解。

可能会包含程序语言理论或其它语言的具体例子作为 语言中，左值指示可能访问的参照。

因为涉及内容过多，本文档不保证内容的完整覆盖，且可能会多次修订。

本文档避免原创研究，尽量保证参考文献可查证但不保证所有文献的权威性。

未完成部分以 TBD(to be determined) 标记。正在修订部分以 WIP(work in progress) 标记。

0 体例说明

0.1 引用文献

仅在此标注权威文献。

正式标准 ISO/IEC 14882:2011 和 ISO/IEC 9899:1999 分别标记为 [C++11] 和 [C11] 。

相关工作组文献，以标准草案为例： ISO/IEC JTC1 WG21 N3936 和 ISO/IEC JTC1 WG14 N1570 ）分别标记为 [WG21/N3936] 和 [WG14/N1570] 。

标记中可包括章节号或章节引用。

引用 ISO C 和 ISO C++ 对应段落的顺序不确定，视文档内容需要而定。

涉及到 ISO C 和 ISO C++ 的差异会特别标示。

引文可能有部分省略。

若无特别说明，保证引用的草案和正式标准中的实质含义一致。

不额外特别标识其它引用。

0.2 记法

和 ISO C 以及 ISO C++ 类似，斜体用于表示首次引入的需要读者注意的局部概念 [C++11 1.3/3] 或语法记法*(syntax notation)* [C++11 1.6] 。这些概念或语法标注在引用的参考文献或本文档内可以找到释义。

粗体用于强调。

0.3 本文的结构

一级目录基本和 ISO C++ 保持一致。

考虑系统性和内容涵盖，在此以 ISO C++ 而不是 ISO C 作为范本。

1 概论

1.1 语法和语义

什么称为语法或语义不是 ISO C 或 ISO C++ 的内容。但是，尽管没有明确定义， ISO C和 ISO C++ 都明确地用到了这个概念。作为背景知识，在这里有必要澄清通常的含义。

所谓语法*(syntax)*，在一般高级语言中指一种描述合规性的“字面上的”构造。其它所有的含义可被归类为语义范畴。

语法和语义不存在绝对的界限。区分语法和语义主要来自于人为的设计。这种差异能够使设计和实现一个语言更简单。从设计的角度来讲，明确语言规则的过程可以首先从提取固定的语法规则开始。语法规则的检查——语法分析具有相对成熟的理论和实现，在此分离这能够让设计者集中于后续的目的性更加明确的语义设计。从实现的角度看，首先检查语法指定的规则，如果不符合则可以直接拒绝接受，避免复杂通用的语义检查造成的实现困难和低效；同时也有机会复用分析器*(parser)* 的实现。

对于没有经过设计的语言比如自然语言来说，语法和语义的界限通常并不明确。和其它形式语言一样，它们也可以用文法*(grammar)* 来描述。

对于显式区分语法和语义的语言来说，语法和语义也可以通过指定形式文法*(formal grammar)* 研究。形式文法被具体表述为记法*(notation)* 时本身构成一种元语言*(metalanguage)*，具有自身的元语法*(meta syntax)* 。元语法的一个经典的例子是 BNF 。

ISO C 和 ISO C++ 使用相似的文法标记。关于体例说明，分别参考 [C11 Clause 6] 和 [C++11 1.6] ；此外，通过 [C11 Annex A] 和 [C++11 Annex A] 总览语法。

此外， ISO C 还对语言规则中使用 Constraints 和 Semantics 为标题进行了分类。

对于实用来说，这里语法和语义的区别重要性，在于它指定了什么程序是应该正确被接受的。

1.2 程序正确性

ISO C++ 定义了合式的*(well-formed)* 程序以及一些其它重要的术语定义：

[WG21/N3936]

1.3 Terms and definitions

1.3.10 [defns.impl.defined]

implementation-defined behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

1.3.12 [defns.locale.specific]

locale-specific behavior

behavior that depends on local conventions of nationality, culture, and language that each implementation documents

1.3.24 [defns.undefined]

undefined behavior

behavior for which this International Standard imposes no requirements

[Note: Undefined behavior may be expected when this International Standard omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. —end note]

1.3.25 [defns.unspecified]

unspecified behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation

[Note: The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this International Standard. —end note]

1.3.26 [defns.well.formed]

> **well-formed program**
>
> C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).

其中：

- 未定义行为*(undefined behavior)* 在 ISO C++11 的意义上理解为“错误的”或“不可移植的”，不保证行为可预测。
- 未指定*(unspecified)* 不保证结果唯一，可以是正确的。
- 实现定义*(implementation-defined)* 类似未指定，但实现有义务在文档上标注唯一确定的选项。
- 可诊断*(diagnosable)* [C++11 Clause 1.4] 详见下文。
- *ODR(One Definition Rule)* [C++11 Clause 3] 独立于一般的语法和语义规则之外，详见下文。

相对地， ISO C 也有类似的术语定义：

[WG14/N1570]

> # 3. Terms, definitions, and symbols
>
> # 3.4
>
> 1 **behavior**
>
> external appearance or action
>
> ## 3.4.1
>
> 1 **implementation-defined behavior**
>
> unspecified behavior where each implementation documents how the choice is made
>
> 2 EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.
>
> ## 3.4.2
>
> 1 **locale-specific behavior**

> behavior that depends on local conventions of nationality, culture, and language that each implementation documents
>
> 2 EXAMPLE An example of locale-specific behavior is whether theislowerfunction returns true for characters other than the 26 lowercase Latin letters.
>
> ## 3.4.3
>
> 1 **undefined behavior**
>
> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements
>
> 2 `NOTE` Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).
>
> 3 `EXAMPLE` An example of undefined behavior is the behavior on integer overflow.
>
> ## 3.4.4
>
> 1 **unspecified behavior**
>
> use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance
>
> 2 `EXAMPLE` An example of unspecified behavior is the order in which the arguments to a function are evaluated.

不同主要在于 ISO C++ 的未指定和实现定义是分离的， ISO C 中的实现定义是未指定的特例。

ISO C 没有规定“合式”，而使用以下机制：

> # 4. Conformance
>
> 1 In this International Standard, “shall” is to be interpreted as a requirement on an implementation or on a program; conversely, “shall not” is to be interpreted as a prohibition.
>
> 2 If a “shall” or “shall not” requirement that appears outside of a constraint or runtimeconstraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words “undefined behavior” or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe “behavior that is undefined”.
>
> 3 A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.
>
> 4 The implementation shall not successfully translate a preprocessing translation unit containing a#errorpreprocessing directive unless it is part of a group skipped by conditional inclusion.

1.3 诊断消息

最典型的诊断消息是编译错误和警告。

一个实现应当正确的程序。对于错误的程序，为了避免实现复杂， ISO C 和 ISO C++ 允许实现不完全进行语义检查。这里略有差异： ISO C 把这种情况统一为未定义行为，而 ISO C++ 特地增加了 no diagnostics required 条款。因此对于 C++ ，编译通过的程序，即便排除未定义行为，在语言规则下仍然可能是错的。

1.4 存储模型

1.5 对象模型

1.5.1 “对象”的正式定义

[WG14/N1570]

> # 3.15
>
> 1 **object**
>
> region of data storage in the execution environment, the contents of which can represent values
>
> 2 `NOTE` When referenced, an object may be interpreted as having a particular type; see 6.3.2.1.

这 ISO C++ 中基本一致：

[WG21/N3936]

> # 1.8 The C++ object model [intro.object]
>
> 1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is a region of storage. [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. —*end note*] An object is created by a *definition* (3.1), by a *new-expression* (5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a *name* (Clause 3). An object has a *storage duration* (3.7) which influences its *lifetime* (3.8). An object has a *type* (3.9). The term *object type* refers to the type with which the object is created. Some objects are *polymorphic* (10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expressions* (Clause 5) used to access them.

在 Java[JLS8] 以及传统的基于类风格的面向对象*(class-based style object-orientation)* 的上下文中，对象则是指“类的实例”，和 ISO C 以及 ISO C++ 都不同。（另外，“实例”的意义在 ISO C++ 中也不一样。）

TBD

1.6 程序执行

1.7 多线程环境

2 词法转换

TBD

3 基本概念

以下“基本概念”指 [C++11 Clause 3] 中讨论的内容。

3.1 名称和实体

ISO C++ 引入 *ODR(One Definition Rule)* [C++11 Clause 3] 而特别指定了实体*(entity)* 的概念。

与此相对，名称*(name)* 用于指称*(denote)* 实体，通过声明*(declaration)* 引入。

推论：名称不是实体。

变量*(variable)* 的含义也在此基础上被明确。

[WG21/N3936]

> # 3 Basic concepts [basic]
>
> 3 An *entity* is a value, object, reference, function, enumerator, type, class member, template, template specialization, namespace, parameter pack, or `this`.
>
> 4 A name is a use of an *identifier* (2.11), *operator-function-id* (13.5), *literal-operator-id* (13.5.8), *conversion-function-id* (12.3.2), or *template-id* (14.2) that denotes an entity or *label* (6.6.4, 6.1).
>
> 5 Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced either by a `goto` statement (6.6.4) or a *labeled-statement* (6.1).
>
> 6 A *variable* is introduced by the declaration of a reference other than a non-static data member or of an object. The variable’s name, if any, denotes the reference or object.

但是， **ISO C** 没有这些概念。 在 ISO C 中：

- vairable 一词并没有被正式使用为名词表示“变量”的含义。在 *VLA(varialbe-length array)* 中出现的是形容词，意为“变量的”，指不在翻译时确定大小，和这里的概念无关。
- entity 一词被不经正式定义地使用，也没有和 ISO C 类似的含义（考虑 **ISO C** 没有 **ODR** ）。似乎作者认为是不言自明的。

- name 一词被不经正式定义地使用。 ISO C 没有 ISO C++ 那样带记号 :: 的 *qualified-id* [C++11 5.1.1] 和 *operator-function-id* [C++11 13.5] 等语法构造，“标识符”和“名称”基本一致。

3.2 声明和定义

首先，预处理 阶段*(phase)* 的 宏定义*(macro definition)* 不属于本章讨论的“定义”的外延。

在 ISO C++ 中，声明总是定义，反过来不成立：

[WG21/N3936]

> # 3.1 Declarations and definitions [basic.def]
>
> 2 A declaration is a *definition* unless it declares a function without specifying the function’s body (8.4), it contains the `extern` specifier (7.1.1) or a *linkage-specification* 25 (7.5) and neither an *initializer* nor a *function-body* , it declares a static data member in a class definition (9.2, 9.4), it is a class name declaration (9.1), it is an *opaque-enum-declaration* (7.2), it is a *template-parameter* (14.1), it is a *parameter-declaration* (8.3.5) in a function declarator that is not the declarator of a *function-definition* , or it is a `typedef` declaration (7.1.3), an *alias-declaration* (7.1.3), a using-declaration (7.3.3), a *static_assert-declaration* (Clause 7), an *attribute-declaration* (Clause 7), an *empty-declaration* (Clause 7), or a *using-directive* (7.3.4).
>
> 25) Appearing inside the braced-enclosed *declaration-seq* in a *linkage-specification* does not affect whether a declaration is a definition.

ISO C 也有类似的结论，但略有不同：

[WG14/N1570]

> # 6.7 Declarations
>
> ## Semantics
>
> 5 A declaration specifies the interpretation and attributes of a set of identifiers. A *definition* of an identifier is a declaration for that identifier that:
>
> — for an object, causes storage to be reserved for that object;
>
> — for a function, includes the function body;119)
>
> — for an enumeration constant, is the (only) declaration of the identifier;
>
> — for a `typedef` name, is the first (or only) declaration of the identifier.
>
> 119) Function definitions have a different syntax, described in 6.9.1.

排除 ISO C++ 不兼容 ISO C 的部分，存在重要的差异：

- **ISO C** 的声明和定义都针对标识符。 **ISO C++** 的声明引入名称（包括声明标识符的情形），而定义针对实体。
- **ISO C** 中的一个声明是否是定义，可能和出现的位置相关。 （例如 *tentative definition* 。）
- `typedef` 声明是可以定义而不总只是声明。

3.3 ODR

3.4 关于变量

上文已经提及 ISO C++ 中的“变量”的定义，且明确了对应术语在 ISO C 中不存在。

ISO C 避免使用“变量”的原因可能在于含义的不够明确。

TBD

与此类似的是，术语“对象”在不同程序语言的规则下也有不同的含义。但是在 **ISO C** 和 **ISO C++** 中对象***(object)*** 的内涵和外延都是清楚的。

3.4.1 “变量”的一般意义

TBD

一般地，变量的“可变”指不同的名称可以绑定不同的实体。

但是，近年来的程序语言设计习惯于曲解为变量可以存储不同的“值”。这是片面的说法。事实上，存在“变量”的值不可变的语言，如 Haskell [Haskell]。

TBD

不管使用什么定义，可以确定的是，变量总有变量名*(variable's name)* 。并且和数学的传统不同，在程序设计语言中，一般不能把两者任意互相替代。

3.4.2 变量和常量

和直觉上不同，在 ISO C++ 的上下文中，变量和常量并不相对：

- ISO C++ 明确定义了什么是变量，但没有定义什么是常量。
- ISO C++中明确涉及常量的术语只有若干常量表达式*(constant-expression)*

常量表达式在实用的意义上指有可能在翻译时即可确定值的表达式，在一般意义上和变量不完全相对。

而在 ISO C 中，却明确有常量*(constant)* 的术语。但是，这只是语法上的分类，相当于 ISO C++ 所说的一部分字面量*(literal)* 。

此外，应该了解，“常量”和下文的 const 限定符是完全两回事，尽管后者在 C++ 中具有常量的目的。

WIP

3.5 作用域

3.6 名称查找

3.7 程序和连接

3.8 启动与终止

3.9 存储期

3.10 对象生存期

3.11 类型

3.11.1 基本类型

ISO C 的基本类型和 ISO C++ 的基本类型不同。

Java 中类似的基本类型称为原始类型*(primitive type)* 。

TBD

3.11.2 组合类型

3.11.3 数组和指针

WIP

数组和指针都是组合类型的分类。数组不是指针，指针不是数组。

不导致歧义时，数组的值或者数组类型的对象可能称为数组。指针类似。同样，在这个上下文中数组不是指针，指针不是数组。

无论何种上下文上面讨论的都是实体。根据名称和实体的推论，数组名不是数组，指针名不是指针。

3.11.4 函数类型

3.11.5 类类型

3.11.6 限定符

3.12 值类别

B 语言指定了左值*(l-value)* 作为语法上的分类。相对地，非左值为右值*(r-value)* 。

C 语言中，左值指示可能访问的存储。右值即为"表达式的值"（参见 [C11 6.3.2.1] 的注释）。除了在特定的一些上下文中，左值会转换为值。

WIP

C++03 规定，表达式不是左值就是右值。

C++11 在引入右值引用类型后引入了新的分类，称为值类别*(value category)* 。左值被扩充为泛左值(glvalue) ，包括原来的左值和新的消亡值(xvalue) ；右值的概念被扩充为包括消亡值和原来的右值，后者称为纯右值(prvalue) 。消亡值同时是泛左值和右值。左值和右值的集合不相交仍然没有改变。此外，一个表达式或者是泛左值，或者是纯右值。

引入消亡值的动机是为了解决涉及右值引用的场合下的表达式的分类。因为这种情况下表达式兼有左值的性质，不适合使用传统的右值，却也不能作为传统的左值。问题具体体现在以下方面([WG21/N3055]) ：

- 右值表示对象本身，而右值引用对应存储中的对象。
- 右值的类型必须是完整的，同时静态类型和动态类型总是一直；右值引用需要有多态性，允许静态类型和动态类型不一致。
- 非类类型的右值不被 cv-qualifier 限定。绑定到 `const` 或 `volatile` 限定对象的右值引用必须保留限定符。
- 右值引用和左值引用一样能绑定到函数。若把返回右值引用的函数调用作为右值，则需要引入新的函数右值，需要较多的改动。

WIP

4 标准转换

4.1 左值到右值转换

4.2 数组到指针转换

数组类型的左值可以转换为指针类型的右值。

4.3 函数到指针转换

4.4 限定符转换

4.5 浮点转换

4.6 整数提升

4.7 通常算术转换

4.8 bool 相关转换

4.9 转换的阶

5 表达式

6 语句

7 声明

8 声明符

9 类

14 模板

概念和纠错

《高质量C++/C编程指南》陷阱

Created @ 2011-04-14, recovered rev 2013-04-12, markdown @ 2015-09-14.

此文硬伤不少，且相对谭XX的书而言隐晦许多，不建议新手学习。 主观的论述，合理的部分，就此略过。原文（点查看）疏漏之处也尽量忍住不吐槽。

第1章 文件结构

> 每个C++/C程序通常分为两个文件。

//错误。没有强调翻译单元的概念。

> 另一个文件用于保存程序的实现（implementation），称为定义（definition）文件。

//有误。实现不简单等同于定义。例如，类的完整声明也是类的定义，但不是完整的实现。头文件也可以存放其它定义（例如模板和内联函数的实现）。

1.2

> ### 建议1-2-1
>
> 在C++语法中，类的成员函数可以在声明的同时被定义，并且自动成为内联函数。这虽然会带来书写上的方便，但却造成了风格不一致，弊大于利。建议将成员函数的定义与声明分开，不论该函数体有多么小。

//这条建议一般不适用于类模板。

1.4

> 头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规则能大大减轻程序员调试、改错的负担。

//在使用合理的情况下，这基本上是正确的，但头文件可以导致其它方面——像重构（典型情况如重命名一个函数）的复杂化。

第2章 程序的版式

这章基本是主观内容。读者需要注意风格是有争议的，其中的“良好风格”或“不良风格”并非公认。

2.8

> 很多C++教课书受到Biarne Stroustrup第一本著作的影响，不知不觉地采用了“以数据为中心”的书写方式，并不见得有多少道理。
>
> 我建议读者采用“以行为为中心”的书写方式，即首先考虑类应该提供什么样的函数。这是很多人的经验——“这样做不仅让自己在设计类时思路清晰，而且方便别人阅读。因为用户最关心的是接口，谁愿意先看到一堆私有数据成员！”

//“Biarne Stroustrup”拼写错误。C++之父的名字是 Bjarne Stroustrup 。有些一厢情愿了：C++ 的类定义中既然允许显式地表示 `private` 成员，就不是纯粹的接口描述方式。以我个人的经验，数据成员写在后面可能会导致顺序阅读代码时需要回溯。当类定义长度较小时这点影响不大，但定义长度比较长的时候（例如 `Loki::Functor` 里的代码）就会需要较频繁地滚屏，影响代码阅读效率。

第3章 命名规则

这章也基本是主观内容。

3.1

> ### 规则3-1-1
>
> 标识符应当直观且可以拼读，可望文知意，不必进行“解码”。

//这点和此文建议使用的匈牙利命名法有一定程度的矛盾。

> ### 规则3-1-3
>
> 命名规则尽量与所采用的操作系统或开发工具的风格保持一致。

//尽管大部分人应该乐于支持这个观点，不过事实上有时候无法实现。例如同时使用标准库和 Windows API 风格的代码。这时倒不妨直接约定允许根据上下文选择要使用的命名风格。要点是，应该让人看出某个名称是用哪个风格命名的，而不至于一眼就混淆来源。

3.2

> 规则**3-2-7**
>
> 为了防止某一软件库中的一些标识符和其它软件库中的冲突，可以为各种标识符加上能反映软件性质的前缀。例如三维图形标准OpenGL的所有库函数均以gl开头，所有常量（或宏定义）均以GL开头。

//在 C++ 中应该考虑是否可以用命名空间代替前缀。

第4章 表达式和基本语句

> 我真的发觉很多程序员用隐含错误的方式写表达式和基本语句，我自己也犯过类似的错误。

//作者似乎没搞清楚“错误”一词的含义。

4.3

> 规则**4-3-4**
>
> 不要写成
>
> ```
>         if (p== 0)  // 容易让人误解p是整型变量
>         if (p!= 0)
> ```

//事实上，C++ 中的 `NULL` 典型地就是 `int` 字面量 `0` （考虑到成文时间，不提新标准的空指针类型），和 `int` 兼容。以 Bjarne Stroustrup 的观点，这样恰恰会使人误以为 `NULL` 不是整数，因此推荐用 `0` 而不是 `NULL` 。

4.3.5

> 或者改写成更加简练的return (condition ? x : y);

//这里的括号是多余的。

4.4

这节不是语言本身而是涉及语言实现的内容。以现在的观点来看，优化器会可能会在此做一些工作。当然了解一些相关原理大体上还是有益的。

第5章 常量

> 常量是一种标识符，它的值在运行期间恒定不变。C语言用 #define来定义常量（称为宏常量）。C++ 语言除了 #define外还可以用const来定义常量（称为const常量）。

//谭XX风格的信口开河。这段引文中逗号或者句号之间的内容，没一个能算得上是正确的。

5.2

> ### 规则5-2-1
>
> 在C++程序中只使用 const常量而不使用宏常量，即const常量完全取代宏常量。

//这是有问题的。事实上很多情况下 const 只能让编译器被修饰的对象当做只读变量，而非编译期的真正意义的常量进行处理。与 #define 的符号常量（字面量）相比，只读变量受到了一些限制，例如不能作 case 的标号。

第6章 函数设计

> C语言中，函数的参数和返回值的传递方式有两种：值传递（pass by value）和指针传递（pass by pointer）。

//错误。形式上， C 语言函数参数只按值传递。所谓的指针传递是按值传递的一种，只是传递参数的类型是指针而已。

6.1

> ### 规则6-1-1
>
> 参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字。如果函数没有参数，则用void填充。

//不妥当。有时候参数的名称并非有意义，像 int max(int, int); 之类的原型，写了也不会让函数的意义更清楚。此外，并没有指出C函数没有参数时参数列表为 void ，这和省略参数列表（接受任意参数）是不同的。而 C++ 中省略参数列表和 void 参数相同，参数列表 ... 接受不确定个数的参数。

6.2

> ### 规则6-2-3
>
> 不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用return语句返回。

//在C++中可以不使用此规则而使用异常（在 C 中理论上也可以类似地使用 setjmp/longjmp ，但容易造成语义不明确，实际上基本不用）。

6.3

> ### 规则6-3-1
>
> 在函数体的“入口处”，对参数的有效性进行检查。

//在函数接口语义明确的情况下并非是必需的。例如 C 标准库 中以及 POSIX 标准中的许多函数。

> ### 规则6-3-2
>
> 在函数体的“出口处”，对return语句的正确性和效率进行检查。

//同样不是必需的。此外检查可能损失效率。

> 要搞清楚返回的究竟是“值”、“指针”还是“引用”。

//注意返回对象的语义，但是刻意区分“值”和“指针”是不必要的，严格上是错误的——它们根本就不是可以比较的一类概念。

> ### 建议6-4-2
>
> 函数体的规模要小，尽量控制在50行代码之内。

//尽管为数不多，有些特殊情况，如编译器的某些分析程序，是明显的反例。

第7章 内存管理

> “640Kought to be enough for everybody
>
> —Bill Gates 1981”

//和本章主题无关。

7.1

> 内存分配方式有三种

//有误。内存分配具体方式由实现决定，语言只限制存储类。

7.2

漏了重复释放内存的错误（这通常会引起程序崩溃）。

> 规则**7-2-1**
>
> 用malloc或new申请内存之后，应该立即检查指针值是否为NULL。防止使用指针值为NULL的内存。

//对 C++ 而言是错误的。 ISO C++ 关于内存分配失败的默认行为是抛出 std::bad_alloc 异常。如果要使分配失败不抛出异常，使用 nothrow 版本，或者设置实现相关的编译选项。

> 规则**7-2-5**
>
> 用free或delete释放了内存之后，立即将指针设置为NULL，防止产生“野指针”。

//不一定必要。例如指针是自动变量，在退出所在的块作用域被自动释放时。

7.3.1

> 该语句企图修改常量字符串的内容而导致运行错误

//有误。 C 语言中字符串字面量（具有数组类型）未必是常量。当然还是应该避免修改字符串字面量，这是未定义行为。

7.9

> 为new和malloc设置异常处理函数。例如VisualC++可以用_set_new_hander函数为new设置用户自己定义的异常处理函数，也可以让malloc享用与new相同的异常处理函数。

//应该补充的是，标准库有 std::set_new_handler 。

第8章 C++函数的高级特性

> const与virtual机制仅用于类的成员函数。

//应该强调“非静态”成员函数，否则就是错误的。

（虽然似乎没有更多明显的错误，不过遗漏的地方一堆，坚决不吐槽……= =）

第9章 类的构造函数、析构函数与赋值函数

9.1

> Stroustrup的命名方法既简单又合理：让构造函数、析构函数与类同名，由于析构函数的目的与构造函数的相反，就加前缀'~'以示区别。

//错误。构造函数和析构函数是无名的（这个细节决定了一些语言特性的限制，例如不能 using 声明基类的构造函数），看起来名称和类名相同的调用方式其实是语法的限制。

> 非内部数据类型的成员对象应当采用第一种方式初始化，以获取更高的效率。

//效率在这里倒是次要的（很容易被编译器优化掉），重要的是语义。另外，初始化列表的一些行为是受到特殊限制的，例如基类子对象和成员的初始化顺序和异常。

9.7

（仍然不想吐槽漏掉 swap & copy 这样高效可靠的方法而在 `operator=` 实现里分配内存……）

第10章 类的继承与组合

首先标题有点问题。类可以继承，组合的应该是类的实例而非本身。

> 如果将对象比作房子，那么类就是房子的设计图纸。所以面向对象设计的重点是类的设计，而不是对象的设计。

//因果关系不成立。而且观点是有问题的，仅适于经典的 class-based OOD 。

10.2

> ### 规则10-2-1
>
> 若在逻辑上A是B的"一部分"（a partof），则不允许B从A派生，而是要用A和其它东西组合出B。

//错误。尽管一般组合能够胜任这项工作，但并非绝对。可以使用private继承完成相同的任务，并且提供覆盖成员函数的特性。这是组合无法完成的。

第11章 其它编程经验

11.1.3

> 如果在编写const成员函数时，不慎修改了数据成员，或者调用了其它非const成员函数，编译器将指出错误

//错误：漏了 `mutable` 修饰的成员这个特例。

> ### 建议11-3-8
>
> 避免编写技巧性很高代码。

//不应该逃避。技巧性高不意味着难以理解；即使难以理解，也可以通过注释弥补。当然，前提是技巧要使用得合理。

> ### 建议11-3-13
>
> 把编译器的选择项设置为最严格状态。

//有时候现实不允许，例如考虑已有代码的兼容性，需要配置时会带来额外的复杂性。

附录C ：C++/C试题的答案与评分标准

> 标准答案示例： const float EPSINON = 0.00001;
>
> if ((x >= - EPSINON) && (x <= EPSINON)

//如果是“标准”的，为什么不用标准库的 `<float.h>` / `<cfloat>` 的 `FLT_EPSILON` 而自己重复发明轮子？

> 三、简答题（25分）
>
> 1、头文件中的ifndef/define/endif 干什么用？（5分）
>
> 答：防止该头文件被重复引用。

//错误。条件编译显然不只用于给头文件增加 header guard 。

> 四、有关内存的思考题（每小题5分，共20分）
>
> 答：程序崩溃。
>
> 因为GetMemory并不能传递动态内存，
>
> Test函数中的 str一直都是 NULL。
>
> strcpy(str, "hello world");将使程序崩溃。

//错误。未定义行为不等同于程序崩溃，尽管很可能是这样。

> 六、编写类String的构造函数、析构函数和赋值函数（25分）
>
> `String& String::operate =(const String &other)`

//关键字 `operator` 拼写错误。实现冗余就不说了。

[科普]字符串和字符串的长度

Created @ 2012-07-30 01:11, recovered rev 2015-09-15, markdown @ 2015-09-16.

首先明确几个概念

字符串：形式语言理论研究的基本对象之一，是字符的有限序列。

形式语言理论研究的基本对象之一，是字符的有限序列。

以下引用中文喂鸡“字符串”：

> 设 `∑` 是叫做字母表的非空有限集合。 `∑` 的元素叫做“符号”或“字符”。在 `∑` 上的字符串（或字）是来自 `∑` 的任何有限序列。例如，如果 `∑ = {0, 1}` ，则 0101 是在 `∑` 之上的字符串。 字符串的长度是在字符串中字符的数目（序列的长度）， 它可以是任何非负整数。“空串”是在 `∑` 上的唯一的长度为0的字符串，并被指示为 `ε` 或 `λ` 。

注意，这里的长度的概念是足够清晰的。

以下引用中文喂鸡“字符串->字符串数据类型”：

> ## 字符串长度
>
> 尽管形式字符串可以有任意（但有限）的长度，实际语言的字符串的长度经常被限制到一个人工极大值。一般的说，有两种类型的字符串数据类型:“定长字符串”，它有固定的极大长度并且不管是否达到了这个极大值都使用同样数量的内存；和“变长字符串”，它的长度不是专断固定的并且依赖于实际的大小使用可变数量的内存。在现代编程语言中的多数字符串是变长字符串。尽管叫这个名字，所有变长字符串还是在长度上有个极限，一般的说这个极限只依赖于可获得的内存的数量。
>
> ……

表示法

一种常用的表示法是使用一个字符代码的数组，每个字符通常占用一个字节（如在 ASCII 代码中）或两个字节（如在 UCS-2 这样的 Unicode 表示中）。它的长度可以使用一个结束符（一般是 NUL， ASCII 代码是 0， 在C编程语言中使用这种方法）。或者在前面加入一个整数值来表示它的长度（在 Pascal 语言中使用这种方法）。

【例略】

可见字符串的长度和存储的关系是不唯一的。

在 C/C++ 中可以使用多种形式表示和存储的字符串。最常见的基本的字符串表示形式（即C标准库/C++标准库都使用的形式）通称为 C 风格字符串(C-style string)，在 ISO C++ 的学名是 NTCTS(null terminated character string) 。

ISO C++11

> ## 17.3.17 [defns.ntcts]
>
> **NTCTS**
>
> a sequence of values that have *character type* that precede the terminating null character type value `charT()`

具体说来，一个典型的场景是：多余一个元素的 `char` / `wchar_t` / `char16_t` / `char32_t` / 其它实现允许的扩展字符类型的数组可以放一个 NTCTS 。

注意，是 a sequence of values 而不是 characters ，表示抽象的含义。下面会看到 character （但不是 multibyte character ）在 C++ 标准库中的明确受限的意义。

顺便，关于 multibyte character 是 C++ 整体通用的基本术语之一，所以独立于 character 之外考虑：

ISO C++11

> ## 1.3.13 [defns.multibyte]
>
> **multibyte character**
>
> sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment
>
> [*Note:* The extended character set is a superset of the basic character set (2.3).—*end note*]

（至于字符、基本执行字符集什么的虽然是必要基础但理解起来很简单，暂且不在此展开。）

ISO C++11

17.5.2.1.4 Character sequences [character.seq]

> 1 The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:
>
> — A *letter* is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.167
>
> — The *decimal-point character* is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clauses 18 through 30 and Annex D by a period, '.', which is also its value in the `"C"` locale, but may change during program execution by a call to `setlocale(int, const char*)`,168 or by a change to a locale object, as described in Clauses 22.3 and 27.
>
> — A *character sequence* is an array object (8.3.4) *A* that can be declared as `T A [N]`, where *T* is any of the types `char`, `unsigned char`, or `signed char` (3.9.1), optionally qualified by any combination of const or volatile. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value *S* that points to its first element.
>
> 167) Note that this definition differs from the definition in ISO C 7.1.1.
>
> 168) declared in `<clocale>` (22.6).

据此可以定义更具体的 NTBS(null terminated byte string)：

ISO C++11

> **17.5.2.1.4.1 Byte strings [byte.strings]**
>
> 1 A *null-terminated byte string*, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the *terminating null* character); no other element in the sequence has the value zero.169
>
> 2 The *length* of an NTBS is the number of elements that precede the terminating null character. An empty ntbs has a length of zero.
>
> 3 The *value* of an NTBS is the sequence of values of the elements up to and including the terminating null character.
>
> 4 A *static* NTBS is an NTBS with static storage duration.170
>
> 169) Many of the objects manipulated by function signatures declared in (21.7) are character sequences or NTBSs. The size of some of these character sequences is limited by a length value, maintained separately from the character sequence.
>
> 170) A string literal, such as "abc", is a static ntbs.

NTBS 的元素通常用 `char` 类型对象或值表示。

NTBS 在 NTCTS 和 character sequence 的基础上明确了存储。此外， NTBS 区分于 NTCTS 的定义的重要目的之一是为了明确（允许变长编码的）多字节字符串 NTMBS 的外延——注意，这里的一些“长度”开始体现出显著的区别。

先看定义：

ISO C++11

> **17.5.2.1.4.2 Multibyte strings [multibyte.strings]**
>
> 1 A *null-terminated multibyte string*, or NTMBS, is an NTBS that constitutes a sequence of valid multibyte characters, beginning and ending in the initial shift state.171
>
> 2 A *static* NTMBS is an NTMBS with static storage duration.
>
> 171) An NTBS that contains characters only from the basic execution character set is also an NTMBS. Each multibyte character then consists of a single byte.

可见 NTMBS 是 NTBS 的子集，它其中可以包含多个（连续）字节组成的字符。

按 17.5.2.1.4.1/2 ， NTMBS 即 NTBS 的长度是其中包含的元素数。这里的“元素”概念和NTBS中有区别，即强调作为 NTMBS 时长度是多字节字符数而不是字符（字节）数。显然对于一般的NTMBS，即便去除结尾的空字符，长度和占用的字节数可以不同。

但是， ISO C 里面关于“长度”可以有些关键性的不同。简而言之，ISO C标准库使用的string相当于 NTBS ，类似 NTMBS 的概念中长度仍以字节计：

ISO C99/C11(N1570)

> 7.1.1/1 A *string* is a contiguous sequence of characters terminated by and including the first null character. The term *multibyte string* is sometimes used instead to emphasize special processing given to multibyte characters contained in the string or to avoid confusion with a wide string. A *pointer to a string* is a pointer to its initial (lowest addressed) character. The *length of a string* is the number of bytes preceding the null character and the *value of a string* is the sequence of the values of the contained characters, in order.

至于宽字符串， ISO C 单独处理（当然长度也是）， 某种意义上从 C 角度来说“宽字符串不是字符串”：

ISO C99/C11(N1570)

> ### 7.1.1 Definitions of terms
>
> 4 A *wide string* is a contiguous sequence of wide characters terminated by and including the first null wide character. A *pointer to a wide string* is a pointer to its initial (lowest addressed) wide character. The *length of a wide string* is the number of wide characters preceding the null wide character and the *value of a wide string* is the sequence of code values of the contained wide characters, in order.

这样对于 `strlen` / `wcslen` 这样的接口来说，所求的长度是非常容易理解的——不包含末尾空字符的字符数；乘上 `sizeof(char)` / `sizeof(wchar_t)` 就是对应不包含末尾空字符的字节数（对于char来说自然可以互换）。

对于多字节字符串， `strlen` （ `string::size` / `string::length` 也一样，反正都不知道编码）的含义就不那么直观了。从和形式抽象相容的定义来讲，NTMBS是恰当的；而NTBS的“长度”只是在多字节表示下的字节数而非字符数。

大概是因为习惯和历史原因， ISO C/C++ 在这里的一致性上做得不太雅观。不过，在未强调编码/字符集前只考虑空字符前的字节数也是可以理解的。

字符串字面量

在 C/C++ 中，还有一个非常被错误理解的特性：字符串字面量(string literal) 。

重点/易错点择要：

1.词法形式

`"xxx"` 、 `L"xxx"` 、 `u"xxx"` 等。另外 raw string literal/user defined literal 以及后者造成的 C++03/C++11 关于一个string literal是否是一个 *token* 的不兼容性等等，这里关系不大，先略过。

2.具有静态存储期。

3.除了 `u` 等起始的 Unicode 字面量，使用的字符集由实现定义（通常由源文件的编码决定）。

因此事实上除了仅包含基本执行字符集中的元素的字符串字面量，相同的输入并不一定导致相同的结果，长度也是由实现定义的。

4.末尾隐含空字符。

5.类型。

对于 C，类型是对应字符类型的数组类型；对于 C++，类型是对应字符类型的 `const` 数组类型。

6.修改引起未定义行为。

7.不等同于字符串。

一个字符串字面量不一定表示一个字符串。

关于最后一点在 ISO C 里有特别指出：

ISO C11(N1570)

> ### 6.4.5 String literals
>
> 6 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.78) ...
>
> 78) A string literal need not be a string (see 7.1.1), because a null character may be embedded in it by a `\0` escape sequence.

例如， `"123\0\0123"` 不是一个字符串，而是两个连续存储的字符串（注意后面的 `\0` 不是一整个字符而是和后面的字符继续构成八进制转义序列）。

除了C风格字符串，还有其它形式。 ISO C++ 标准库提供了类模版 `std::basic_string` 。对于 C++03 ， 它不一定以 NTCTS 的形式存储；而对于 C++11 ， 因为接口的限制，它的实例对象在可观察行为上表现为内部存储一个NTCTS。

非标准库的形式也有很多。比如说 M$ 的 `_bstr_t` ；或者自己实现一个类 PASCAL 字符串，等等。

对于这类非标准字符串，如果涉及到多字节字符且不明确实现（通常来说对于用户不应该明确实现）的话，通常应当约定“长度”指的是字符数还是字节数。

至于数组长度，完全不上道的玩意儿。

也就是字符串字面量用来初始化数组约定隐式数组的长，和C风格字符串都没什么直接关系。

新手指引：

Appended @ 2012-7-30 01:32.

- 明确一般意义形式上的字符串和C风格字符串的差异；
- 明确 C 风格字符串和字符串字面量的差异；
- 明确空字符不是字符串的内容；
- 明确一般意义上的字符串长度和字符集/编码相关；
- 明确 C 风格字符串的长度通常只讨论不包括末尾空字符的字节数。

剩下基本问题不大了。

非标准库字符串进阶参考：

Appended @ 2012-7-30 01:39.

- C++1y 候选的 `string_ref` ；
- LLVM 中的各种字符串表示形式的实现 。

面向对象和所谓的“面向过程”

Created @ 2012-10-10, v3 rev 2012-10-12, markdown @ 2014-11-07.

摘要

本文综述面向对象，尤其是面向对象编程的基本概念和一些其它编程范型的比较，并指出了现有初学者的一些常见误区。

I.面向对象(Object-Oriented, OO)综述

公认的面向对象是一种“思想”，更精确地说是一种方法学*(methodology)* 。面向对象编程*(OOP)* 、面向对象分析*(OOA)* 、面向对象设计*(OOD)* 等范畴是对此的衍生。容易理解，OOP 指使用 OO 的方法进行编程； OOA 和 OOD 分别指使用 OO 的方法进行系统分析与设计（可以合称 OOAD ），是 OO 方法学在软件工程上的应用。对于使用 OO 方法学的软件开发， OOP 是基础也是具有更强的普遍性（即便使用 OO 方法，也并非所有的软件都有必要使用系统化的 OOA 和 OOD 进行开发），通过 OOP 可以反映 OO 在编程实践中的重要应用，因此本文着重论述 OOP 的有关内容。关于 OOAD ，读者可以在了解OOP的基础上自行学习。

II.编程范型(programing paradigm)

是计算机编程中的一种基本方式[en.wiki:programming paradigm] 。 OOP 和命令式编程*(imperative programing)* 、函数式编程*(functional programing)* 、 逻辑编程*(logical programing)* 并列，是当前主流的编程范型[Kurt Nørmarks]。此外还有结构化*(structured)* 、声明式*(deriective)* 、面向方面*(aspect-oriented)* 、数据驱动*(data-driven)* 、泛型*(generic)* 、并行*(parallel)* 、元编程*(metaprograming)* 等各种范型。

应该注意的是，这些范型并不都是同一层次上的风格，且由于分类方法的不同，不都是互斥的。当然也有些范型是对立的：结构化与非结构化*(non-structured)* ，但这是少数。因此通常在同一段程序中使用了一种以上的编程范型，只强调其中的一部分。

III.结构化编程(structued programing)、命令式编程(imperative programing)和过程式编程(procedural programing)

早期的程序没有强调任何范型，是非结构化的。结构化程序由子程序的执行、选择、迭代构成，无需跳转。

命令式编程是一种重要的编程范型。它和声明式编程相对，强调特定路径执行的步骤（控制流）使程序的状态向预期改变，而非和执行路径无关的计算逻辑的表达。大部分硬件实现的体系结构都直接支持命令式范型。可执行的语句作为语言特性，是支持命令式编程的重要特征。

过程式编程有时被看作是命令式编程的同义词，也可以表示一种基于结构化编程和过程调用(procedual call)的编程范型[en.wiki:Procedural programming]。过程（或例程、子例程、方法、函数——注意不要和函数式编程混淆）在这里是可以通过调用这一手段被重复执行的程序片段，作为语言特性，是支持过程式编程的语言的重要特征。

下文中提及的过程式编程都指第二种含义，而第一种直接称为命令式编程。

无论是命令式还是过程式的编程范型都被许多编程语言广泛支持。对于Pascal、C、C++、Java等语言，两者都是最基本的（使用时几乎无法避免的）范型，而结构化结构化通过这两个范型的上层被体现。

IV.作为编程范型(programing paradigm)的OOP

OO 程序可以看成一系列对象的交互，而不是如传统的过程式编程那样执行一系列任务（过程 ）。

如 OO 字面所说，对象*(object)* 是其中的一个要素。

OOP 中每个对象都有能力接收、处理和向其它对象发送消息*(message)* ，可以被看作具有不同角色或职责的独立单元。对象的动作（或“方法*(method)* ”）与之紧密相关。例如，OOP 数据结构倾向于携带自身的操作（或至少从类似的对象或类“继承 ”）。

对于过程式编程，程序可以被分解为若干过程。 OOP 的做法不同——它分解程序为若干对象和对象的交互（尽管其中的“方法”仍然可以扮演过程式编程中的过程的角色）。

传统的过程式编程中过程对数据的访问是不加外部限制的，而OOP的做法不同——它使用访问权限控制等特性，强调封装，鼓励程序员使某一部分的数据仅可被特定的程序片段（过程）访问，以减少可能发生的错误。

对象和消息是 OOP 的核心。 OOP 和对类型系统的抽象导致类*(class)* 概念的出现。使用类作为核心特性的语言实现 OOP 的基于类的风格的*(class-based style)* OO ，这是支持OOP的语言中的主流，如 C++ 、 Java 。与之不同的是基于原型的风格的*(prototype-based style)* OO，如ECMAScript。

当前流行的 C++ 、 Java 等“主流面向对象语言”（事实上把 C++ 称为面向对象语言并太不恰当，见下文），其实并不能算最符合 OOP 的基本要义。原始的 OOP 的观点，见面向对象之父 Alan Kay 的阐述[Alan Kay] 。

OOP 常具有如下几个特性：数据抽象、封装、消息、组件化、多态和继承。注意，这些特性并非同时提出的，因此并非都是 OOP 的必要组成部分；也并非 OOP 的专利。

应该指出的是，即便不使用直接的 OOP 特性，也可以进行 OOP 。例如 C 可以使用结构体模拟类。

V.语言相关的实例：Java和C++的OOP和其它范型的支持策略的比较

有些语言为特定的范型设计，鼓励用户使用特定的范型。如 Java 鼓吹的“简单”的“纯” OOP 。但是通过上文可以知道， Java 支持的面向对象只是一种基于类的风格的 OOP ，并且血统远非纯净。另外， Java 不支持多重（实现）继承而只支持接口继承，即便仅从基于类风格的 OOP 上来说也有缺陷（由于这点，加上 Java 缺乏其它一些有用的特性，这给混入 *(mixin)* 等带来麻烦，也容易造成代码冗长）。语言构造也决定 Java 无法摆脱命令式和过程式编程，在这个意义上并不“纯”。（尽管 Java 支持泛型，但功能过于薄弱，在此忽略。）使用好 Java 需要用户对 OOP 的较清晰全面的理解，事实上一点都不“简单”。

另外一些语言不强调（甚至弱化）特定的范型，鼓励用户选择合适的范型，如 C++ 。

关于 OOP 两者还有一些整体上重要的、原则性的不同。 Java 的 OOP 特性支持的设计试图减少 OOP 和 OOAD 的差距，使 OOAD 的结果尽量能直接 OOP 对应。初衷可以理解，但效果不见得好（取决于用户设计的合理性），反而几乎肯定增大了语义上的复杂性——这点的一个例子是“继承”不考虑 private 成员。而 C++ 的设计事实上无视这一点。 C++ 的 class type 并不见得就是 OOAD 意义上的 class ， 它可以是类似 Java 里的 interface 或者和 OOP 无关的东西——如元编程用到的 traits 。这点可能导致学习上的困难，但增大了灵活性。

VI.结论：一些需要避免的误区

首先，所谓“面向过程”并不是公认的编程范型的名称，无法和过程式、 OOP 等相提并论。提出这个生造的术语可能只是效仿“面向对象”的构词，却忽略了其中的内涵。通常使用这个术语通常似乎是想表达“过程式编程”。在已有更精确和广泛接受的术语的情形下，不应该使用这种模糊的称谓。

其次，关于语言和范型。尽管语言可以直接通过语言特性支持范型，但范型实质上是跟语言无关的。尤其对于 C++ 而言，强调“支持面向对象”作为和其它语言的主要区别，是没有根据的。

参考文献

[en.wiki:*] 英文喂鸡，喂度娘，略。

[Kurt Nørmarks]

[Alan Kay]Dr. Alan Kay on the Meaning of “Object-Oriented Programming”

关于 main 函数的原型和返回值

Created @ 2012-11-07.

发现以前说的太零碎，不太好引用……整理一下。

目前我看到的比较靠谱的说法（有正确的引用出处，并指出了实现扩展）：
http://homepage.ntlworld.com/jonathan.deboynepollard/FGA/legality-of-void-main.html

http://tieba.baidu.com/p/626323902

↑而这里的说法是有问题的。

这里再解释一下ISO C/C++中对main的要求。

0 标准版本说明

基本内容参照术语和文献列表。

在本文问题上， ANSI C89 和 C90 、 C99 和 C11 、 C++ 标准各个版本这三组标准之间分别没有实质变化（或根本一模一样），所以只引用最早的标准文本。

1 首先是几个背景知识

本文所讲的实现即语言实现，可以是编译器+链接器等等，可以是解释环境。一般是前者。 关于 implementation-defined 等确切含义可以 Google 。

1.1.ANSI C89支持函数声明省略返回值，隐含为 `int`

也就是说 `main()` 其实是 `int main()` ， `foo();` 其实是 `int foo();` 。尤其注意 `main()` 绝不是 `void main()` 。

这在 ISO C99 开始以及 ISO C++ 中是不允许的。

1.2 关于参数列表

C语言的 `(void)` 或函数定义中的 `()` 表示不接受任何参数，相当于 C++ 的 `()` ，也和 C++ 的 `(void)` 等价。

C 语言的 `()` 在函数定义外表示接受任何参数，相当于 C++ 的 `(...)` 。

但是，声明中的空参数列表（非原型声明）“ `()` ”的使用是过时用法（容易造成误会），不建议使用。

所以在 C 语言中，声明时最好不要省略 `(void)` 中的 `void` ，要是省略就不是预期想要的函数原型了。在定义中可以使用 `()` ，如 `int main(){}` ，同 `int main(void){}` 。但若要保证声明和定义通用，只用 `(void)` 表示函数没有参数。

而 C++ 中，不接受任何参数的参数列表写成 `(void)` 是不必要的（虽然也没错，但正式写法都没有这种无谓的罗嗦）。

相关依据：

ISO C11(N1570)

> **6.7.6.3**/14 An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.145)
>
> 145) See "future language directions" (6.11.6).
>
> # 6.11.6 Function declarators
>
> 1 The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

1.3 实现环境分类

ISO C/C++ 中，根据对环境的要求，分为两类，一类是独立实现*(freestanding implementation)*，另一类是宿主实现*(hosted implementation)*。

独立实现对环境的要求比较低，所以更自由。宿主实现——一般可以看作是有操作系统的实现，提供了比较多的底层接口，约束比较多。

当然 C 和 C++ 之间对两者的要求有所不同。为简化问题，除了 `main` 相关的部分在下文讨论以外，不再提及。

1.4 ISO 标准文档中的情态动词的含义

以下全部节录（供参考，只想看结论的可以跳过）。

表格项分隔使用 | ，同一格内不同项使用 / 。

> # ISO/IEC Directives, Part 3 Annex E(normative)
>
> ## Verbal forms for the expression of provisions
>
> `NOTE` Only singular forms are shown.
>
> The verbal forms shown in Table E.1 shall be used to indicate requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted.
>
> ### Table E.1 — Requirement
>
> Verbal form Equivalent expressions for use in exceptional cases(see 6.6.1.3)
>
> shall | is to/is required to/it is required that/has to/only … is permitted/it is necessary
>
> shall not | is not allowed [permitted] [acceptable] [permissible]/is required to be not/is required that … be not/is not to be
>
> Do not use “must” as an alternative for “shall”. (This will avoid any confusion between the requirements of a standard and external statutory obligations.)
>
> Do not use “may not” instead of “shall not” to express a prohibition. To express a direct instruction, for example referring to steps to be taken in a test method, use the imperative mood in English.
>
> `EXAMPLE` “Switch on the recorder.”
>
> The verbal forms shown in Table E.2 shall be used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others, or that a certain course of action is preferred but not necessarily required, or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.
>
> ### Table E.2 — Recommendation
>
> Verbal form Equivalent expressions for use in exceptional cases(see 6.6.1.3)
>
> should | it is recommended that/ought to
>
> should not | it is not recommended that/ought not to

> In French, do not use “devrait” in this context.
>
> The verbal forms shown in Table E.3 are used to indicate a course of action permissible within the limits of the standard.
>
> ## Table E.3 — Permission
>
> Verbal form Equivalent expressions for use in exceptional cases(see 6.6.1.3)
>
> may | is permitted/is allowed/is permissible
>
> need not | it is not required that/no … is required
>
> Do not use “possible” or “impossible” in this context.
>
> Do not use “can” instead of “may” in this context.
>
> `NOTE 1` “May” signifies permission expressed by the standard, whereas “can” refers to the ability of a user of the standard or to a possibility open to him.
>
> `NOTE 2` The French verb “pouvoir” can indicate both permission and possibility. For clarity, the use of other expressions is advisable if otherwise there is a risk of misunderstanding.
>
> The verbal forms shown in Table E.4 are used for statements of possibility and capability, whether material, physical or causal.
>
> ## Table E.4 — Possibility and capability
>
> Verbal form Equivalent expressions for use in exceptional cases(see 6.6.1.3)
>
> can | be able to/there is a possibility of/it is possible to
>
> cannot | be unable to/there is no possibility of/it is not possible to
>
> `NOTE` See note 1 to Table E.3.

这里最需要注意的有两点：

- shall 的意义，以及区分 shall 和 should 。
- shall 表示要求， should 表示建议。这点举个例子，在 ISO C++ 关于容器的 `size()` 的复杂度上应该能坑到一些人。

ISO C++98/03 在表格的 note 里要求 “should” O(1) ，因此 libstdc++ 的 `std::list::size()` 可以实现为 O(n) 的，类似 `std::distance(l.begin(), l.end())` ，不违反标准要求。

而 ISO C++11 改成了 “shall” O(1) ，这种实现就不符合标准了。

can 的含义。

应该注意 can 表示可能性，而不是要求。和表示准许的may也应该有清楚的区别。

2 正题

附一个参考链接：http://stackoverflow.com/questions/1765686/correctly-declaring-the-main-function-in-ansi-c

2.1 ANSI C/ISO C 对独立环境的规定

ANSI C89 是这样的：

> ### 2.1.2.1 Freestanding environment
>
> In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. There are otherwise no reserved external identifiers. Any library facilities available to a freestanding program are implementation-defined.
>
> The effect of program termination in a freestanding environment is implementation-defined.

可见独立环境中不要求有 `main` 函数存在作为入口函数，更不限定 `main` 的原型。

ISO C99 是这样的：

> ### 5.1.2.1 Freestanding environment
>
> 1 In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. Any library facilities available to a freestanding program, other than the minimal set required by clause 4, are implementation-defined.
>
> 2 The effect of program termination in a freestanding environment is implementation-defined.

没什么变化。

ISO C++98 是这样的：

> ### 3.6.1 Main function [basic.start.main]
>
> 1 A program shall contain a global function called main, which is the designated start of the program. It is implementation-defined whether a program in a freestanding environment is required to define a main function. [Note: in a freestanding environment, startup and termination is implementation-defined; startup contains the execution of constructors for objects of namespace scope with static storage duration; termination contains the execution of destructors for objects with static storage duration.]

2.2 ANSI C89 对宿主环境的规定

> ### 2.1.2.2 Hosted environment
>
> A hosted environment need not be provided, but shall conform to the following specifications if present.
>
> "Program startup"
>
> The function called at program startup is named main . The implementation declares no prototype for this function. It can be defined with no parameters:
>
> `int main(void) { /*...*/ }`
>
> or with two parameters (referred to here as argc and argv , though any names may be used, as they are local to the function in which they are declared):
>
> `int main(int argc, char *argv[]) { /*...*/ }`
>
> If they are defined, the parameters to the main function shall obey the following constraints:
>
> - The value of argc shall be nonnegative.
> - `argv[argc]` shall be a null pointer.
> - If the value of argc is greater than zero, the array members argv[0] through `argv[argc-1]` inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both upper-case and lower-case, the implementation shall ensure that the strings are received in lower-case.
> - If the value of argc is greater than zero, the string pointed to by argv[0] represents the program name ; `argv[0][0]` shall be the null character if the program name is not available from the host environment. If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the program parameters .
> - The parameters `argc` and `argv` and the strings pointed to by the argv array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.
>
> ...

对宿主环境可能有的(can)：有 `main` ，并且 `main` 的原型是这里提到的其中之一。 若使用第二种原型，参数有一定限制(shall)。 `argc` 必须非负， `argv[argc]` 是空指针。 `argv[0]` ... `argv[argc-1]` 表示程序参数。一般实现中支持使用命令行传递这些参数。

由于这里是 can，实际上也允许其它原型。

但这种说法显然太隐晦了。

2.3 ISO C99 对宿主环境的规定

> ## 5.1.2.2 Hosted environment
>
> 1 A hosted environment need not be provided, but shall conform to the following specifications if present.
>
> #### 5.1.2.2.1 Program startup
>
> 1 The function called at program startup is named `main`. The implementation declares no prototype for this function. It shall be defined with a return type of int and with no parameters:
>
> ```
> int main(void) { /* ... */ }
> ```
>
> or with two parameters (referred to here as argc and argv, though any names may be used, as they are local to the function in which they are declared):
>
> ```
> int main(int argc, char *argv[]) { /* ... */ }
> ```
>
> or equivalent;9) or in some other implementation-defined manner.
>
> 2 If they are declared, the parameters to the main function shall obey the following constraints:
>
> — The value of `argc` shall be nonnegative.
>
> — `argv[argc]` shall be a null pointer.
>
> — If the value of `argc` is greater than zero, the array members `argv[0]` through `argv[argc-1]` inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.
>
> — If the value of argc is greater than zero, the string pointed to by `argv[0]` represents the program name; `argv[0][0]` shall be the null character if the program name is not available from the host environment. If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the program parameters.
>
> — The parameters `argc` and `argv` and the strings pointed to by the argv array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

这里实质的变化是明确要求 ANSI C 中的两种原型必须被宿主实现接受。而 or in some other implementation-defined manner 的妥协可以看成是对 can 的兼容。

2.4 ISO C++98 的规定

> # 3.6 Start and termination [basic.start]
>
> ## 3.6.1 Main function [basic.start.main]
>
> 1 A program shall contain a global function called main, which is the designated start of the program. It is implementation-defined whether a program in a freestanding environment is required to define a main function. [Note: in a freestanding environment, startup and termination is implementation-defined; startup contains the execution of constructors for objects of namespace scope with static storage duration; termination contains the execution of destructors for objects with static storage duration.]
>
> 2 An implementation shall not predefine the main function. This function shall not be overloaded. It shall have a return type of type int, but otherwise its type is implementation-defined. All implementations shall allow both of the following definitions of main:
>
> `int main() { /* ... */ }`
>
> and
>
> `int main(int argc, char* argv[]) { /* ... */ }`
>
> In the latter form argc shall be the number of arguments passed to the program from the environment in which the program is run. If argc is nonzero these arguments shall be supplied in `argv[0]` through `argv[argc1]` as pointers to the initial characters of nullterminated multibyte strings (NTMBSs)(17.3.2.1.3.2) and `argv[0]` shall be the pointer to the initial character of a NTMBS that represents the name used to invoke the program or `""`. The value of argc shall be nonnegative. The value of `argv[argc]` shall be 0. [Note: it is recommended that any further (optional) parameters be added after `argv`.]
>
> 3 The function main shall not be used (3.2) within a program. The linkage (3.5) of main is implementation-defined. A program that declares main to be inline or static is illformed. The name main is not otherwise reserved. [Example: member functions, classes, and enumerations can be called main, as can entities in other namespaces.]

ISO C++ 的规定和 ISO C 类似，但有几点重要的不同：

- a)程序必须包含全局 `main` 。但在独立实现中不一定用到 `main` 作为程序启动的入口，事实上程序启动过程是由实现定义的。
- b)实现不能预定义 `main` （以免用户定义的 `main` 冲突违反 one definition rule 导致错误）。注意这不和上文矛盾。例如，链接器可以在生成可执行二进制映像时决定是否应

  该添加 `main` 的定义。
- c)main必须返回 `int` 。
- d)全局 `main` 禁止被使用。因此不像 C， C++ 中 `main` 无法递归调用。 `&::main` 也是错误的。

3 结论

3.1 `main` 的兼容性

别盲目认为哪个是对的哪个是错的，标准没这么简单。

`void main()` 在 C 仍然可以是符合标准*(conforming)* 的扩展，只要有文档——看我一开始给的链接。只不过依赖 implementation-defined 不是 strictly conforming 罢了，可移植性较弱。

在 C++ 中不返回 `int` 的 `main` 直接不符合标准。

3.2 基于保证可移植性的入口函数使用的建议策略

以下不适用于自己实现语言或者写操作系统之类。

- a)确定实现。如果是独立实现，自己翻实现文档，否则
- b)如无特殊必要，尽量使用标准明文规定的两种形式；
- c)使用其它形式应能找到文档，并且确保当前需求能容忍由此导致的可移植性缺陷。

附

ISO C99 起，及 ISO C++98 起，全局 `main` 若没有 `return` ，相当于末尾隐含 `return 0;` 。对于一般实现，返回 0 表示程序执行成功。 C/C++ 标准库宏 `EXIT_SUCCESS` 表示由实现定义的成功返回状态。 `EXIT_SUCCESS` 可用于 `exit` 函数，而 `main` 终止和 `exit` 语义上等价，所以也可以 `return EXIT_SUCCESS;` 代替。

C11 & C++11 的赋值相关表达式求值

Created @ 2013-01-09, markdown @ 2014-11-09.

警告：本文需要有入门级的语言常识和一定的语言理论基础。

1.关于求值(evaluation)

C99/C++03 虽然正式地使用到这个词，但没有明确具体的外延。

C11/C++11 则把 evaluation 作为局部术语给出了明确了的定义：

ISO C11(N1570)

> ### 5.1.2.3
>
> 2 访问一个 volatile 对象, 修改一个对象, 一个文件, 或者调用一个函数， 这些操作都算副作用, 副作用是执行环境状态的变化. 表达式的求值一般包括值的计算(both value computations)和引发副作用. 一个左值表达式的值计算包括确定左值所指示的对象的同一性(determining the identity of the designated object).

ISO C++11

> ## 1.9
>
> 12 通过 volatile 泛左值(glvalue)(3.10)访问一个指定对象 , 修改一个对象, 调用一个库IO函数, 或者调用一个函数， 这些操作都算副作用, 副作用是执行环境状态的变化. Evaluation of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access to a volatile object is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the volatile access may not have completed yet.

即求值在一般意义上包括值的计算(value computations)和副作用的产生(initiation of side effects)。其中左值（C++中glvalue同之前的左值lvalue）的求值包含确定左值所指示的对象的同一性。

2.可观察行为(observable behavior)

上面有一点需要特别注意： `volatile` 对象的读取也是副作用。 `volatile` 的根本含义根本体现在对可观察行为的影响，而可观察行为则约定抽象语义和现实实现中的程序语义的联系（这事实上决定了一个符合标准的实现最多能通过修改程序语义优化到的界限）：

ISO C11(N1570)

> 1 The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.
>
> 6 The least requirements on a conforming implementation are:
>
> — Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
>
> — At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.
>
> — The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input. This is the observable behavior of the program.

ISO C++11

> ## 1.9
>
> 1 The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.5
>
> 5) This provision is sometimes called the “as-if” rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.
>
> 8 The least requirements on a conforming implementation are:
>
> — Access to volatile objects are evaluated strictly according to the rules of the abstract machine.
>
> — At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.
>
> — The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.
>
> These collectively are referred to as the observable behavior of the program. [Note: More stringent correspondences between abstract and actual semantics may be defined by each implementation. —end note]

3.求值的顺序

和 Java 、 C# 等不同，对于 C 和 C++ 来说，表达式的求值顺序和字面上（从左到右）没有一致对应关系，也就是说优先级、结合性等纯粹是语法上的现象，不决定语义。

求值的顺序由额外的规则指定。原则上若不能决定能得到可预测的结果的，则指定为未定义行为*(undefined behavior)*，对结果不做出任何保证和要求。这在 C99 和 C++03 都有一定的体现，都用到了序列点*(sequence point)* 的概念：

ISO C99

> ### 5.1.2.3
>
> 2 Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects,11) which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place. (A summary of the sequence points is given in annex C.)
>
> ## 6.5
>
> 2 Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.70)
>
> 70) This paragraph renders undefined statement expressions such as

`i = ++i + 1;`

`a[i++] = i;`

while allowing

`i = i + 1;`

`a[i] = i;`

ISO C++03

> ## 1.9
>
> 7 Accessing an object designated by a volatile lvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression might produce side effects. At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.7)
>
> 7) Note that some aspects of sequencing in the abstract machine are unspecified; the preceding restriction upon side effects applies to that particular execution sequence in which the actual code is generated. Also note that when a call to a library I/O function returns, the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) may not have completed yet.

而在C++11中，根据提案WG21/N1944，序列点被先序(sequenced before)二元关系等术语取代。WG21/N2052在此基础上（另一个基础是约定内存模型的WG21/N2052）阐明了并发的内存模型——关于适合多线程环境下的特定并发语义，本文从略。　　尽管ISO C看起来起先并不热心，最后还是接受了这个修正，但在ISO C11（至少是N1570）中保留了序列点的说法。（另外ISO C++11似乎遗漏了sequenced after，尽管从下文可以看出，在赋值的定义中这个词组正式地被使用。）

WG21/N1944

> According to my best guesses, there are at least two factors which have inhibited progress in the C committee:
>
> > WG14 seems collectively to believe that the problem is not urgent or compelling; that most implementers and experts understand the situation well enough to agree on the essentials of the requirements, and that most of the confusion can be treated as an education problem (not within the committee's scope).
> >
> > Most of the proposals to clarify matters are so radical that they are (rightly) met with considerable suspicion, and (accurately) judged to require much deep consideration to ensure that they don't constitute an unacceptable change to the status quo.

ISO C11(N1570)

> ## 5.1.2.3
>
> 3 Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is sequenced after A.) If A is not sequenced before or after B, then A and B are unsequenced. Evaluations A and B are indeterminately sequenced when A is sequenced either before or after B, but it is unspecified which.13) The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B. (A summary of the sequence points is given in annex C.)
>
> 13) The executions of unsequenced evaluations can interleave. Indeterminately sequenced evaluations cannot interleave, but can be executed in any order.

ISO C++11

> 13 Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.10), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are unsequenced. [Note: The execution of unsequenced evaluations can overlap. —end note] Evaluations A and B are indeterminately sequenced when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [Note: Indeterminately sequenced evaluations cannot overlap, but either could be executed first. —end note]

同时，限定求值顺序的条款也有大幅度的修改：

ISO C11(N1570)

> # 6.5
>
> 1 An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.
>
> 2 If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.84)
>
> 84) This paragraph renders undefined statement expressions such as

`i = ++i + 1;`

`a[i++] = i;`

while allowing

`i = i + 1;`

`a[i] = i;`

ISO C++11

> 14 Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.8.

> 15 Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. [Note: In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. —end note] The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a scalar object is unsequenced relative to either anotherside effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. [Example:
>
> ```
> void f(int, int);
> void g(int i, int* v) {
> i = v[i++]; // the behavior is undefined
> i = 7, i++, i++; // i becomes 9
> i = i++ + 1; // the behavior is undefined
> i = i + 1; // the value of i is incremented
> f(i = -1, i = -1); // the behavior is undefined
> }
> ```
>
> —end example]
>
> When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. [Note: Value computations and side effects associated with different argument expressions are unsequenced. —end note] Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.9 Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [Example: Evaluation of a new expression invokes one or more allocation and constructor functions; see 5.3.4. For another example, invocation of a conversion function (12.3.2) can arise in contexts in which no function call syntax appears. —end example] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.
>
> 9) In other words, function executions do not interleave with each other.

注意到 ISO C++11 的篇幅大了很多，是因为它把原先不少 ISO C++03 属于第 5 章的内容提前到第 1 章来讲了。可见这里的普遍性和重要性。

这里的修改最大的整体差异是对于求值中各个过程的粒度的细化。序列点的描述把值的计算和副作用之间的顺序捆绑在一起，而先序关系可以更清晰地分别描述这两类需要明确顺序的过程之间的逻辑上保持的顺序关系（至于对于整个执行环境来说的顺序，由于要考虑并发，由 *happens before* 二元关系等精确描述，这里从略）。

4.ISO C11/ISO C++11的修正

接下来需要了解的就是各个具体的表达式的求值顺序，这里从略。

重点是 ISO C11/ISO C++11 对 ISO C99/ISO C++03 做了语义上的修订，在明确求值顺序规则的依赖的基础定义（先序关系）的同时，放宽了部分条件，导致原来存在未定义行为的表达式现在可以是正确的。　　首先的例子是 WG21 Core Defect Report 637， 关于上面 ISO C++11最终使用的例子。

> ## 637. Sequencing rules and example disagree
>
> ## Section: 1.9 [intro.execution] Status: CD1 Submitter: Ofer Porat Date: 2 June 2007
>
> [Voted into the WP at the September, 2008 meeting.]
>
> In 1.9 [intro.execution] paragraph 16, the following expression is still listed as an example of undefined behavior:
>
> ```
> i = ++i + 1;
> ```
>
> However, it appears that the new sequencing rules make this expression well-defined:
>
> > The assignment side-effect is required to be sequenced after the value computations of both its LHS and RHS (5.17 [expr.ass] paragraph 1).
> >
> > The LHS (i) is an lvalue, so its value computation involves computing the address of i.
> >
> > In order to value-compute the RHS (++i + 1), it is necessary to first value-compute the lvalue expression ++i and then do an lvalue-to-rvalue conversion on the result. This guarantees that the incrementation side-effect is sequenced before the computation of the addition operation, which in turn is sequenced before the assignment side effect. In other words, it yields a well-defined order and final value for this expression.
>
> It should be noted that a similar expression

> ```
> i = i++ + 1;
> ```
>
> is still not well-defined, since the incrementation side-effect remains unsequenced with respect to the assignment side-effect.
>
> It's unclear whether making the expression in the example well-defined was intentional or just a coincidental byproduct of the new sequencing rules. In either case either the example should be fixed, or the rules should be changed.
>
> Clark Nelson: In my opinion, the poster's argument is perfectly correct. The rules adopted reflect the CWG's desired outcome for issue 222. At the Portland meeting, I presented (and still sympathize with) Tom Plum's case that these rules go a little too far in nailing down required behavior; this is a consequence of that.
>
> One way or another, a change needs to be made, and I think we should seriously consider weakening the resolution of issue 222 to keep this example as having undefined behavior. This could be done fairly simply by having the sequencing requirements for an assignment expression depend on whether it appears in an lvalue context.
>
> James Widman: How's this for a possible re-wording?
>
> > In all cases, the side effect of the assignment expression is sequenced after the value computations of the right and left operands. Furthermore, if the assignment expression appears in a context where an lvalue is required, the side effect of the assignment expression is sequenced before its value computation.
>
> **Notes from the February, 2008 meeting:**
>
> There was no real support in the CWG for weakening the resolution of issue 222 and returning the example to having undefined behavior. No one knew of an implementation that doesn't already do the (newly) right thing for such an example, so there was little motivation to go out of our way to increase the domain of undefined behavior. So the proposed resolution is to change the example to one that definitely does have undependable behavior in existing practice, and undefined behavior under the new rules.
>
> Also, the new formulation of the sequencing rules approved in Oxford contained the wording that by and large resolved issue 222, so with the resolution of this issue, we can also close issue 222.
>
> **Proposed resolution (March, 2008):**
>
> Change the example in 1.9 [intro.execution] paragraph 16 as follows:

```
i = v[i++];             // the behavior is undefined
i = 7, i++, i++;        // i becomes 9
i = 【去掉++i】 i++ + 1;        // the behavior is undefined
i = i + 1;              // the value of i is incremented
```

这里很明显地，认为 `i = ++i + 1` 不再存在求值顺序不确定的未定义行为。原因是 `++i` 依赖的 += 的求值顺序有修改。

这个修改的理由同样由 Core Defect Report 提出：

> ## 222. Sequence points and lvalue-returning operators
>
> ## Section: 5 [expr] Status: CD1 Submitter: Andrew Koenig Date: 20 Dec 1999
>
> [Voted into the WP at the September, 2008 meeting.]
>
> I believe that the committee has neglected to take into account one of the differences between C and C++ when defining sequence points. As an example, consider
>
> ```
> (a += b) += c;
> ```
>
> where `a`, `b`, and `c` all have type `int`. I believe that this expression has undefined behavior, even though it is well-formed. It is not well-formed in C, because `+=` returns an rvalue there. The reason for the undefined behavior is that it modifies the value of `a' twice between sequence points.
>
> Expressions such as this one are sometimes genuinely useful. Of course, we could write this particular example as
>
> ```
> a += b; a += c;
> ```
>
> but what about
>
> ```
> void scale(double* p, int n, double x, double y) {
>     for (int i = 0; i < n; ++i) {
>         (p[i] *= x) += y;
>     }
> }
> ```
>
> All of the potential rewrites involve multiply-evaluating p[i] or unobvious circumlocations like creating references to the array element.

> One way to deal with this issue would be to include built-in operators in the rule that puts a sequence point between evaluating a function's arguments and evaluating the function itself. However, that might be overkill: I see no reason to require that in
>
> ```
> x[i++] = y;
> ```
>
> the contents of ``i`' must be incremented before the assignment.
>
> A less stringent alternative might be to say that when a built-in operator yields an lvalue, the implementation shall not subsequently change the value of that object as a consequence of that operator.
>
> I find it hard to imagine an implementation that does not do this already. Am I wrong? Is there any implementation out there that does not `do the right thing' already for (a += b) += c?
>
> 5.17 [expr.ass] paragraph 1 says,
>
> > The result of the assignment operation is the value stored in the left operand after the assignment has taken place; the result is an lvalue.
>
> What is the normative effect of the words "after the assignment has taken place"? I think that phrase ought to mean that in addition to whatever constraints the rules about sequence points might impose on the implementation, assignment operators on built-in types have the additional constraint that they must store the left-hand side's new value before returning a reference to that object as their result.
>
> One could argue that as the C++ standard currently stands, the effect of `x = y = 0;` is undefined. The reason is that it both fetches and stores the value of `y`, and does not fetch the value of y in order to compute its new value.
>
> I'm suggesting that the phrase "after the assignment has taken place" should be read as constraining the implementation to set `y` to `0` before yielding the value of `y` as the result of the subexpression `y = 0`.
>
> Note that this suggestion is different from asking that there be a sequence point after evaluation of an assignment. In particular, I am not suggesting that an order constraint be imposed on any side effects other than the assignment itself.
>
> Francis Glassborow:
>
> My understanding is that for a single variable:
>
> Multiple read accesses without a write are OK

> A single read access followed by a single write (of a value dependant on the read, so that the read MUST happen first) is OK
>
> A write followed by an actual read is undefined behaviour
>
> Multiple writes have undefined behaviour
>
> It is the 3) that is often ignored because in practice the compiler hardly ever codes for the read because it already has that value but in complicated evaluations with a shortage of registers, that is not always the case. Without getting too close to the hardware, I think we both know that a read too close to a write can be problematical on some hardware.
>
> So, in `x = y = 0;`, the implementation must NOT fetch a value from `y`, instead it has to "know" what that value will be (easy because it has just computed that in order to know what it must, at some time, store in `y`). From this I deduce that computing the lvalue (to know where to store) and the rvalue to know what is stored are two entirely independent actions that can occur in any order commensurate with the overall requirements that both operands for an operator be evaluated before the operator is.
>
> Erwin Unruh:
>
> C distinguishes between the resulting value of an assignment and putting the value in store. So in C a compiler might implement the statement `x=y=0;` either as `x=0;y=0;` or as `y=0;x=0;` In C the statement `(x += 5) += 7;` is not allowed because the first += yields an rvalue which is not allowed as left operand to `+=`. So in C an assignment is not a sequence of write/read because the result is not really "read".
>
> In C++ we decided to make the result of assignment an lvalue. In this case we do not have the option to specify the "value" of the result. That is just the variable itself (or its address in a different view). So in C++, strictly speaking, the statement `x=y=0;` must be implemented as `y=0;x=y;` which makes a big difference if `y` is declared volatile.
>
> Furthermore, I think undefined behaviour should not be the result of a single mentioning of a variable within an expression. So the statement `(x +=5) += 7;` should NOT have undefined behaviour.
>
> In my view the semantics could be:
>
> if the result of an assignment is used as an rvalue, its value is that of the variable after assignment. The actual store takes place before the next sequence point, but may be before the value is used. This is consistent with C usage.

> if the result of an assignment is used as an lvalue to store another value, then the new value will be stored in the variable before the next sequence point. It is unspecified whether the first assigned value is stored intermediately.
>
> if the result of an assignment is used as an lvalue to take an address, that address is given (it doesn't change). The actual store of the new value takes place before the next sequence point.
>
> Jerry Schwarz:
>
> My recollection is different from Erwin's. I am confident that the intention when we decided to make assignments lvalues was not to change the semantics of evaluation of assignments. The semantics was supposed to remain the same as C's.
>
> Ervin seems to assume that because assignments are lvalues, an assignment's value must be determined by a read of the location. But that was definitely not our intention. As he notes this has a significant impact on the semantics of assignment to a volatile variable. If Erwin's interpretation were correct we would have no way to write a volatile variable without also reading it.
>
> Lawrence Crowl:
>
> For `x=y=0`, lvalue semantics implies an lvalue to rvalue conversion on the result of y=0, which in turn implies a read. If `y` is volatile, lvalue semantics implies both a read and a write on `y`.
>
> The standard apparently doesn't state whether there is a value dependence of the lvalue result on the completion of the assignment. Such a statement in the standard would solve the non-volatile C compatibility issue, and would be consistent with a user-implemented `operator=`.
>
> Another possible approach is to state that primitive assignment operators have two results, an lvalue and a corresponding "after-store" rvalue. The rvalue result would be used when an rvalue is required, while the lvalue result would be used when an lvalue is required. However, this semantics is unsupportable for user-defined assignment operators, or at least inconsistent with all implementations that I know of. I would not enjoy trying to write such two-faced semantics.
>
> Erwin Unruh:
>
> The intent was for assignments to behave the same as in C. Unfortunately the change of the result to lvalue did not keep that. An "lvalue of type int" has no "`int`" value! So there is a difference between intent and the standard's wording.
>
> So we have one of several choices:

> live with the incompatibility (and the problems it has for volatile variables)
>
> make the result of assignment an rvalue (only builtin-assignment, maybe only for builtin types), which makes some presently valid programs invalid
>
> introduce "two-face semantics" for builtin assignments, and clarify the sequence problematics
>
> make a special rule for assignment to a volatile lvalue of builtin type
>
> I think the last one has the least impact on existing programs, but it is an ugly solution.
>
> Andrew Koenig:
>
> Whatever we may have intended, I do not think that there is any clean way of making
>
> ```
> volatile int v;
> int i;
>
> i = v = 42;
> ```
>
> have the same semantics in C++ as it does in C. Like it or not, the subexpression `v = 42` has the type ``reference to volatile int,'' so if this statement has any meaning at all, the meaning must be to store 42 in v and then fetch the value of v to assign it to i.
>
> Indeed, if v is volatile, I cannot imagine a conscientious programmer writing a statement such as this one. Instead, I would expect to see
>
> v = 42; i = v;
>
> if the intent is to store 42 in `v` and then fetch the (possibly changed) value of `v`, or
>
> ```
> v = 42;
> i = 42;
> ```
>
> if the intent is to store `42` in both `v` and `i`.
>
> What I do want is to ensure that expressions such as ` ``i = v = 42 `'' have well-defined semantics, as well as expressions such as `(i = v) = 42` or, more realistically, `(i += v) += 42` .
>
> I wonder if the following resolution is sufficient:
>
> Append to 5.17 [expr.ass] paragraph 1:
>
> > There is a sequence point between assigning the new value to the left operand and yielding the result of the assignment expression.

> I believe that this proposal achieves my desired effect of not constraining when j is incremented in x[j++] = y, because I don't think there is a constraint on the relative order of incrementing j and executing the assignment. However, I do think it allows expressions such as (i += v) += 42, although with different semantics from C if v is volatile.
>
> **Notes on 10/01 meeting:**
>
> There was agreement that adding a sequence point is probably the right solution.
>
> **Notes from the 4/02 meeting:**
>
> The working group reaffirmed the sequence-point solution, but we will look for any counter-examples where efficiency would be harmed.
>
> For drafting, we note that `++x` is defined in 5.3.2 [expr.pre.incr] as equivalent to `x+=1` and is therefore affected by this change. `x++` is not affected. Also, we should update any list of all sequence points.
>
> **Notes from October 2004 meeting:**
>
> Discussion centered around whether a sequence point “between assigning the new value to the left operand and yielding the result of the expression” would require completion of all side effects of the operand expressions before the value of the assignment expression was used in another expression. The consensus opinion was that it would, that this is the definition of a sequence point. Jason Merrill pointed out that adding a sequence point after the assignment is essentially the same as rewriting
>
> ```
> b += a
> ```
>
> as
>
> ```
> b += a, b
> ```
>
> Clark Nelson expressed a desire for something like a “weak” sequence point that would force the assignment to occur but that would leave the side effects of the operands unconstrained. In support of this position, he cited the following expression:
>
> ```
> j = (i = j++)
> ```
>
> With the proposed addition of a full sequence point after the assignment to i, the net effect is no change to j. However, both g++ and MSVC++ behave differently: if the previous value of j is 5, the value of the expression is 5 but j gets the value 6.

> Clark Nelson will investigate alternative approaches and report back to the working group.
>
> **Proposed resolution (March, 2008):**
>
> See issue 637.

最大的问题出在赋值上。

5.赋值中的求值顺序

对于赋值的修订的要义可以直接通过对照简单赋值的标准条款的差异看出来。复合赋值的等价展开形式和求值一次的规则并没有变化。

ISO C99

> ## 6.5.16
>
> 3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point.
>
> 4 The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined.

ISO C++03

> # 5.17
>
> 1 There are several assignment operators, all of which group right-to-left. All require a modifiable lvalue as their left operand, and the type of an assignment expression is that of its left operand. The result of the assignment operation is the value stored in the left operand after the assignment has taken place; the result is an lvalue.

ISO C11(N1570)

> ### 6.5.16
>
> 3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment,111) but is not an lvalue. The type of an assignment expression is the type the left operand would have after lvalue conversion. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

111) 暂略。

ISO C++11

> ## 5.17
>
> 1 The assignment operator (`=`) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue referring to the left operand. The result in all cases is a bit-field if the left operand is a bit-field. In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression. With respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. [Note: Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single compound assignment operator. —end note]

概括起来，明显的变化是现在可以确定对操作数的值的计算、更新左操作数和整个赋值表达式的值的计算依序进行。当操作数的求值是赋值（以及语义同 `+= 1` 的内建前缀 `++` 时），这里的副作用和更新包含它的赋值表达式的左操作数这个副作用之间的顺序就被确定了——这是之前没有的。

因此，对于内建非 `volatile` 类型， `i = ++i + 1` 这个表达式不再具有未定义行为。

这个修改的可行性在于是放宽而不是加强限制，且现有的实现基本无需做出修正就可以直接符合新的标准要求。

5. C 和 C++ 在内建赋值上的差异

无关上面的修正，有一个根本差异都没有变化： C 的赋值表达式不是左值， C++ 的赋值表达式是左值。

这在判断涉及 `volatile` 左值的赋值时应该尤为注意。

对于 C 来说，赋值得到一个值而不是 `volatile` 修饰的左值，作为左操作数时不涉及必然读 `volatile` 对应的存储：

ISO C11(N1570)

> 111) The implementation is permitted to read the object to determine the value but is not required to, even when the object has volatile-qualified type.

这里若需要读取，只能理解为赋值自身的副作用。

事实上因为赋值表达式不是左值，也不可能允许 `(x = 1) = 2` 这样的表达式。所以从属是确定的，不存在顺序不明确的问题。

但对于C++来说，情况就不一样了。作为右操作数的 `volatile` 左值需要通过 lvalue-to-rvalue conversion 转换为右值，这是对 `volatile` 的读取的额外副作用。左操作数的左值若需要保持左值则不需要。 这里还有个陷阱：若赋值表达式是完全表达式，那么整个表达式作为左值是否应该转换成右值（对 `volatile` 类型来说导致一次额外副作用）？ 这个问题在C++03中的答案是右值，也就是说对 `volatile` 左值赋值后需要无条件从被赋值中左值取得存储的值来作为右值。这在实现中可以被轻易优化掉，但在语言规则中存在一定问题，如：

```
void f()
{
    volatile int x;
    x; // Undefined behavior?
}
```

若坚持这一点，上面的代码就存在未定义行为，因为读取未初始化的对象——即便读出来的值实际并没有被用到。 这看起来不太合理。 Core Defect Report 举出了类似的例子（虽然没涉及未定义行为，从中也可以看出 C 和 C++ 在这个问题上的差异）：

> # 1054. Lvalue-to-rvalue conversions in expression statements
>
> ## Section: 6.2 [stmt.expr] Status: FDIS Submitter: Hans Boehm Date: 2010-03-16
>
> [Voted into the WP at the March, 2011 meeting.]
>
> C and C++ differ in the treatment of an expression statement, in particular with regard to whether a volatile lvalue is fetched. For example,
>
> ```
> volatile int x;
> void f() {
>     x;    // Fetches x in C, not in C++
> }
> ```
>
> The reason C++ is different in this regard is principally due to the fact that an assignment expression is an lvalue in C++ but not in C. If the lvalue-to-rvalue conversion were applied to expression statements, a statement like
>
> ```
> x = 5;
> ```
>
> would write to x and then immediately read it.
>
> It is not clear that the current approach to dealing with the difference in assignment expressions is the only or best approach; it might be possible to avoid the unwanted fetch on the result of an assignment statement without giving up the fetch for a variable appearing by itself in an expression statement.
>
> **Proposed resolution (January, 2011):**

略，见下文。

C++11 最终采纳了这个建议，增设一段，明确保证不转换：

ISO C++11

> **5**
>
> 10 In some contexts, an expression only appears for its side effects. Such an expression is called a discarded-value expression. The expression is evaluated and its value is discarded. The array-to-pointer (4.2) and functionto- pointer (4.3) standard conversions are not applied. The lvalue-to-rvalue conversion (4.1) is applied only if the expression is an lvalue of volatile-qualified type and it has one of the following forms:
>
> — id-expression (5.1.1),
>
> — subscripting (5.2.1),
>
> — class member access (5.2.5),
>
> — indirection (5.3.1),
>
> — pointer-to-member operation (5.5),
>
> — conditional expression (5.16) where both the second and the third operands are one of the above, or
>
> — comma expression (5.18) where the right operand is one of the above.

这个概念同时也用于简化其它一些规则的描述。

[科普]变量、全局变量及其它

Created @ 2013-01-28, v2 rev 2013-01-29, markdown @ 2015-09-14.

对最近各种蛋疼的所谓“全局”“变量”问题的解释做个小结。

1.变量(variable) 的一般含义

“变量”来源于代数学，是数学中最伟大的发明之一。变量是表示可变数学对象的符号(symbol) 。它具有两重含义，一是指在某个上下文中的符号本身；二是这个符号表示的可变的值(value) 。

2.程序设计语言中的变量、变量名(variable name) 和作用域(scope)

在程序设计语言中也有类似的概念。不同的是，一般需要更加明确地指出一个变量有效的上下文。这个上下文通常是代码中的一段（连续的）区域(region) ，称为作用域(scope) 。只有在作用域中使用的变量是有效的。

由于作用域针对的是一段代码区域，所以对于变量的两重含义的约束是不同的。作用域实际只限定变量作为符号本身，也就是字面形式，即变量名(variable name) 。在语言的语法中，通常标识符(identifier) 这一成分就可以作为变量名。当然变量名也可以是更复杂的表达式。

而变量的另一种含义，即变量的内容，各种语言可以提供不同的抽象，例如可以直接指定存储，也可以约定使用其它抽象的语义形式如何体现可变的状态。

另外，在明确了名称的概念之后，作用域也不只限于变量，对于任何其它具有名称的抽象也适用。作用域能避免相同的名称的冲突，也就是说允许相同的名称在不同的上下文中指称(denotes) 不同的内容。

3.限定名称(qualified name)

上面提到过使用作用域的一个原因是防止相同名称的冲突。有时作用域限制不太方便，因为需要在整个程序范围内访问的名称都得放在全局作用域中。所以还需要其它的机制。

许多语言提供了全局名称的额外组织机制，一个比较通用的手段是允许对标识符增加表示作用域子集的前缀构成限定名称。限定名称更加明确，而对应未被限定的名称则可以适合更多作用域。

这种机制可以有不同的表现形式，如C++和C#的命名空间(namespace)，Ada 和 Java 的包(pakage) 等（Java 首先把变量放在类中，然后以类作为包的成员）。

4.C语言的变量

应该指出， K&R 提到了变量，而 ISO C 中没有正式定义这个概念，甚至 ISO C 正式文本中用到的 variable 一词也都不是变量的意思，而是如 variable length array 等。

这里的变量可以按传统意义理解，即包括变量名在内，但对于 C 这样的明确支持不同作用域的语言来说，在整个程序范围内变量的同一性(identity) 就成了问题——“变量名一致的变量就是同一个变量”有违整体上的直觉。而要是撇开造成问题的变量名，即专指变量内容，在 C 语言中使用表示存储的对象(object) 就能很好地解释清楚了，没必要用“变量”这个词来增加理解上的困难。不知是不是这个原因，ISO C 才没定义变量的概念。

也就是说，对变量的概念存在两种理解，不见得哪种就是对的，而且都没有必要。这种二义性已经使得一些基本问题的讨论变得没有意义，如“变量是不是表达式”。由于讨论语言问题时“变量”可以被更清晰的术语取代，因此，可以回避这个模糊的说法。

5.C++语言的变量

C++ 中的变量通过对象或不是作为类的非静态成员的引用的声明引入。变量名指称被声明的引用或对象：

ISO C++11

> **3**/6 A variable is introduced by the declaration of a reference other than a non-static data member or of an object. The variable’s name denotes the reference or object.

由于引用的存在，这里变量的概念并没有被对象等直接取代。

6.全局(global)

全局是指整个程序的范围。例如，对于运行时来说，全局状态是程序的各个部分都能访问的状态。

全局变量(global variable) 是指程序代码中的各个作用域都能访问的变量。但是，下面会看到，C/C++代码无法真正地达到这点。

与全局对立的是局部(local)。与全局变量对立的是局部变量(local variable)。但是，C/C++程序中，局部变量往往指块作用域(block scope) 中的变量，并不严格对立。因此下文不使用这些概念。

7.C语言的作用域和名称空间

C语言有且仅有函数作用域(function scope)、文件作用域(file scope)、块作用域和函数原型作用域(function prototype scope) 这些作用域：

ISO C11(N1570)

> ## 6.2.1 Scopes of identifiers
>
> 3 A label name is the only kind of identifier that has function scope. It can be used (in a goto statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a : and a statement).
>
> 4 Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has file scope, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has block scope, which terminates at the end of the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has function prototype scope, which terminates at the end of the function declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the inner scope) will end strictly before the scope of the other entity (the outer scope). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is hidden (and not visible) within the inner scope.

顺带纠正两点常见误区：

- a)函数体内中声明的名称所在的作用域是块作用域，而不是函数作用域，后者是标号(label) 专有的（这条也适用于C++）。
- b)没有所谓的全局作用域，通常所谓的全局充其量只是在文件作用域而已。

不过在C语言中同一作用域其实允许有限地允许不同实体重名。除了限定名称外，C 语言还有一种消除歧义的方式——名称空间(name space) ：

ISO C11(N1570)

> ## 6.2.3 Name spaces of identifiers
>
> 1 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate name spaces for various categories of identifiers, as follows:
>
> — label names (disambiguated by the syntax of the label declaration and use);
>
> — the tags of structures, unions, and enumerations (disambiguated by following any32) of the keywords struct, union, or enum);
>
> — the members of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the . or -> operator);
>
> — all other identifiers, called ordinary identifiers (declared in ordinary declarators or as enumeration constants).
>
> 32) There is only one name space for tags even though three are possible.

C语言没有限定名称，所有的名称都是标识符。ISO C也没有直接说明“名称”的含义，但这里应该是清楚的。（实际上，C语言语法中的identifier在其前身B语言中的对应物就叫name 。）

8.C++语言的作用域：

C++的作用域比较多，以下以标准文本的标题排列：

ISO C++11

- 3.3.3 Block scope [basic.scope.local]
- 3.3.4 Function prototype scope [basic.scope.proto]
- 3.3.5 Function scope [basic.funscope]
- 3.3.6 Namespace scope [basic.scope.namespace]
- 3.3.7 Class scope [basic.scope.class]
- 3.3.8 Enumeration scope [basic.scope.enum]
- 3.3.9 Template parameter scope [basic.scope.temp]

注意到块作用域一节的交叉引用(cross reference) 的标签(label) 中说的是“local”。这是因为 C++98/03 中用词比较混乱，有时候作 block scope 有时候作 local scope（局部作用域），到 C++11 按 ISO C 统一了，但交叉引用在没有整节作废时还是需要保持兼容性，所以没变。

C++ 的名称是指标识符或限定标识符的使用：

ISO C++11

> **3**/4 A name is a use of an identifier (2.11), operator-function-id (13.5), literal-operator-id (13.5.8), conversion-function-id (12.3.2), or template-id (14.2) that denotes an entity or label (6.6.4, 6.1).

关于“全局”，这里需要提一点：

ISO C++11

> **3.3.6**/3 The outermost declarative region of a translation unit is also a namespace, called the global namespace. A name declared in the global namespace has global namespace scope (also called global scope). The potential scope of such a name begins at its point of declaration (3.3.2) and ends at the end of the translation unit that is its declarative region. Names with global namespace scope are said to be global name.

也就是说 C++ 有“全局（命名空间）作用域”，是命名空间作用域的特例，其中名称为全局名称。可以看出它和C的文件作用域是对应的。

（为什么 C 没有真正的“全局”而 C++ 可以有了呢——并不是这个单独决定的。）

9.C/C++程序与链接(linkage)

一个C/C++程序由一个或多个翻译单元(translation unit) 组成。翻译单元作为源代码可以各自独立地被翻译为目标代码然后链接(linking) 成完整的程序。语法上，（预处理后的、正确的）翻译单元由名称的声明(declaration) 构成。声明引入标识符/名称并确定它们的指称。某个翻译单元中的声明仅在这个翻译单元内有效。合法的程序中使用任意名称之前都需要这个名称在所在翻译单元内（而不是“全局”）的（用户提供的，或者实现预定义的）声明。

翻译单元的地位是等价的；不存在“全局”的翻译单元。因此，一般地，C/C++ 程序不存在源码层次上的“全局”的概念：“任意名称”自然也包括变量名，只能依赖于具体翻译单元。这样，不管是 C 还是 C++ ，都无所谓真正意义上的“全局变量”。（考虑到 C90 允许隐式声明，倒是有些“全局”的风格；但这种手法不被当前的 C/C++ 接受。）

但语言允许在不同翻译单元中共享程序的全局状态——可以直接通过共享不同翻译单元的实体的指称实现。只要指定不同翻译单元中文件/全局作用域中的某些名称指称相同的对象就可以达到“全局变量”的目的。相应地，也存在和外部无关，只是指称翻译单元内部存在的实体名称。这样的属性称为链接(linkage) 。注意链接是针对名称的，而不是对象等具体实体。

很自然地，能以相同名称共享外部实体的名称具有外部链接(external linkage) ，只在翻译单元内部共享指称实体的具有内部链接(interal linkage) ，不共享指称的无链接(no linkeage) 。

所谓外部变量(external variable) 就是指变量名具有外部链接的变量。这和作用域没有直接的关系。但由于对象声明默认具有的链接，容易造成混淆。这里仅举两例，不详细展开：a) `extern` 的确切含义；b) `const` 对象在 C 和 C++ 中具有不同的链接。

这样，可以确定，外部变量和"全局变量"不是一回事。这方面的误解看来还是不少，不知道拜谁所赐了。

10.结论：C/C++中"全局变量"的确切含义

可见，无论是"全局"还是"变量"，在 C 和 C++ 之间都有一些差距。

所谓"全局"，无论在 C/C++ 中，都没有传统的意义。 C 能实现在效果上和"全局变量"类似的只是名称具有外部链接的对象，但硬说"全局"就名不副实了。 C++ 多了全局命名空间这种在源码层次上强制的约定，但要真正能保证实现"全局"，还是得靠链接。

C 语言中讨论"全局变量"可以有各种没明确的意义，所以这个混乱的概念还是不用为妙。只有C++中是可以明确的：指全局命名空间作用域中的变量。"文件作用域对象"才是C语言中对应的比较明确的提法。

C 的这种混乱的根源除了 ISO C 没有明确一些和传统认知有微妙差异的关键概念外，主要是由于缺乏对通过链接共享实体指称的多翻译单元的语言设计的理解所致。不过说到底，既然是坑，没能力填上就绕过去吧。

关于异常处理的一些话题

Created @ 2013-03-09, v4 rev 2013-03-27, markdown @ 2015-09-14.

摘要

本文对C++语言提供的异常处理机制和异常安全问题进行简要的介绍，并明确关于异常机制在应用中遇到的若干问题及其解决方法，最后讨论异常规范以及C++11对此的改动。

关键字

异常处理，异常安全，RAII，异常中立，异常规范。

Abstract

This article briefly talked about exception handling mechanism provided by C++ and exception safety, and clarifies some problems and solution dealing with error handling. It also concerns exception specification and its evolution in C++11.

Keywords: C++, exception handling, exception safety, RAII, exception neutrality, exception specification

I.绪论：问题边界

若无其它说明，本文讨论的异常处理特指C++的一项重要的特性。本文关注它适合的应用范围，但不讨论具体语言实现细节。

关于异常处理的基本语法和相关语言特性的使用细节，参见参考资料和其它相关教材。

II.背景：异常和异常处理

异常在计算机系统中被认为是非正常状态的抽象。异常能够干预程序的正常控制流，它很大程度地影响程序在某种预期条件以外的表现[1]。

一般地说，当一个异常被处理(handled)时，程序控制流切换至称为异常处理器(exception handler)的子例程中。若异常是可继续(continuable)的，程序可以利用先前保存的信息切换回正常控制流。

异常处理被实现为硬件机制和特定的软件构造。前者的例子有IEEE浮点异常[1]、x86架构的双重故障(double fault)异常[2]；后者如Windows结构化异常处理(SEH)[1]，以及在多种语言如C++、Java、Ada等被内建支持[1][3]。

对于软件实现的异常处理机制来说，术语“异常”典型地被作为储存异常条件(exception condition)的数据结构。异常的传输在这里有普遍的相似性：产生(raise)或抛出(throw)一个异常中断正常控制流，直至被捕获(catch)而处理。若一个异常能在程序的任意部分产生，那么这样的异常是异步(asynchronous)的，否则是同步(synchronous)的。

由程序设计语言提供支持的异常一般通过编译器生成代码和运行时库动态的检查来实现。编译器需要在确定在特定位置生成代码，这里异常的异步性是受限的。因此，保留某些异常不由语言的实现而是由底层的操作系统或硬件提供支持是合理的。

这样，不同层次上的异常处理机制处理不同的异常。例如，在安装Windows上的PC运行（带有运行时异常支持的）标准C++程序，浮点数异常被硬件处理、影响环境的某些状态后可继续执行；整数除以零或内存访问越界由Windows包装为结构化异常并抛出；通过new操作符分配失败时默认抛出std::bad_alloc异常。被抛出的异常若不被用户显式处理，默认情况下导致程序最终非正常退出。

III.C++异常

可见不是所有的程序逻辑的异常都被C++处理。标准C++只处理同步异常。

一些实现可能支持扩展，如VC++支持非标准关键字**try**、except等Windows SEH特性。对于可移植的C++程序，不应使用这些特性。

通行的做法是，当遇到需要抛出的异常条件时，用一个多态类(polymorphic class)类型的对象（是一种临时对象，称为异常对象）来储存，然后通过throw抛出，到最终需要处理异常处使用和try块对应的catch捕获。一般为了方便异常对象的生存期管理，catch指定的类型为异常对象的引用（const在此会被忽略，没有必要）。构造异常对象时一般应避免产生新的异常。

`catch(...)` 可以捕获任意类型的异常对象。需要注意的是，在上述带有扩展的实现中，catch(...)可能不安全地捕获到非C++异常[4]。所以通常避免使用抛出任意类型的异常以及catch(...)，而使用自己的异常类继承体系，以保证具有可移植性同时能确保适当情况下捕获所有异常[4]。

此外，C++可以使用异常规范(exception specification)来约束一个函数（函数模版）是否接受异常，或接受特定类型的异常，这在本文最后讨论。

IV.异常安全

异常安全是指在抛出异常后保持可预期的状态。通过Abrahams异常安全保证描述异常安全性[4-6]：

基本保证——允许失败操作改变程序状态，但不能有泄漏并且失败操作所影响的对象/模块必须仍然在生存期内并可用，状态必须是可靠的(consistent)（但不是完全可预测的）。

强保证——包括事务式的提交/撤销语义：失败操作必须保证关于该操作的对象的程序状态不被改变。

无抛出保证——根本不会发生失败操作，该操作不会抛出异常。

获得异常安全的一般原则：

使用“资源获取初始化”("resource acquisition is initialization")(RAII)来管理资源的分配——在析构函数中释放资源——自动对象会在异常抛出时析构，释放资源无需用户干预；

“从小处做好所有工作，然后保证只使用无异常抛出的操作”从而避免改变程序内部状态，直到能保证整个操作成功；

坚持“一个类（或函数），只做一个任务”。

参照标准库的策略保证异常安全：一个函数应当总是支持最严格的保证，同时不会对不需要这种保证的用户造成伤害。

注意，一些关键函数，如析构函数和去配(deallocation)函数，必须是无抛出保证的操作，否则在某些条件（具体来说，抛出异常时的栈回退）下无法避免未定义行为，导致程序行为无法预测[3]。

V.异常、失败(failure)和错误(error)

异常表示非正常，它蕴含了失败——确定在不能完成接口约定的功能且无法在程序中恢复的情形。异常不一定是失败，而失败是异常（虽然有时候为了强调非失败的异常而单独提取出来）。

错误（程序设计意义上的，不是指软件的bug）是程序员需要关注处理的对象，包括失败和违反接口约束但可能恢复的异常。用契约式设计(design by contract)来概括，这里的接口约束包含三个方面：前置条件(precondition)、不变量(invariant)和后置条件(postcondition)[1][7]。

错误不应该经常发生。经常发生的状况应该被预期，而不是作为错误处理(error handling)的对象[7]。

对于某些可恢复的异常，C++标准库提供了一些错误恢复例程，如对于默认new失败时[7]首先会调用new handler，无法恢复时才抛出异常。用std::set_new_hanlder等标准库函数来设置这样的例程[3]。

VI.错误处理的方式

错误处理是C++异常处理最典型的应用领域。虽然语言并不阻止没有错误的流程中使用异常，但这样可能会导致代码不易读或导致调试时过于容易中断，往往被视为滥用。

不使用异常处理的典型错误处理手段是静态存储状态和传递错误码(error code)。这两种方案都有其局限性。

静态存储错误状态（可能存储的就是错误码，如POSIX的errno、Win32的GetLastError）的局限性很明显；

需要静态地分配空间——意味着要么允许错误状态被覆盖（这强迫用户必须在可能发生错误后第一时刻检查错误状态），要么浪费空间（不管可能产生错误的函数是否被调用到都需要）；

在多线程环境下需要考虑同步及额外开销（若使用线程局部存储，则对实现的要求比较高）；

几乎是无法重入(reenterable)的。

错误码通过函数参数或返回值传递（如Win32的GetLastError），每一次手动转发只能跨一层函数，和抛出、捕获异常相比，这导致一些明显不利[7]：

冗余——显式转发随调用层次增加而递增；

混淆正常流程(happy path)和错误处理流程——往往需要很多代码判断错误码，错误处理代码和其它代码没有清晰的边界；

非健壮性——错误码处理实现有误时程序携带错误状态执行，难以确定实现的缺陷会如何暴露；

难以确保和检查正确性——若要保证正确，每一处都需检查所有可能的错误是否被处理；

耦合——若修改错误码，每一处判断都需要重新检查是否需要修改；

无法用于泛型代码——错误码不具备对类型编码的能力，以至于无法区分不同实例类型中相同代码表示的不同错误[7]；

无法简洁地跨不同调用层次统一处理错误。

特别地，使用返回值传递错误码有如下无法克服的复杂和困难：

必须预留返回值给错误——可能占用原本语义上合理的返回值；

只能保存极其有限的状态——返回值只有一个；

使嵌套调用更复杂——多余的参数需要额外的显式声明；

需要小心区分不同返回值的含义；

在没有可用的返回值时——构造函数、重载、转换函数中根本无法使用。

异常相对于错误码还有其它优势：携带足够的信息，按异常类的继承体系汇总不同错误统一处理，类型安全（不会被莫名其妙地转换）、容易被特定的工具检查等[7]。

有观点认为使用C++异常有显著的性能负担，因此应该尽量使用错误码处理错误。这是不正确的。现代编译器可以做到不抛出异常时没有时间开销；相比之下空间开销通常不成问题[7]。但是，应当避免过于频繁地抛出和捕获异常。

因此如有可能，应该尽量选择异常作为错误处理机制，除了以下情况：

优势无法发挥——错误处理和产生错误的地点非常接近时；

造成性能问题——通常是不恰当地使用（例如把频繁出现的状况作为错误）导致的。

VII.使用异常进行错误处理的策略

这里需要解决的问题是：什么时候抛出/捕获/不抛出也不捕获（使异常隐式地向上层调用者传递，即异常中立）异常？

首先是关于未确定为不可恢复的错误时的恢复策略。若确定可恢复时，可以直接原地恢复（例如配置文件不存在，就创建配置文件）；否则重新抛出异常，交给上层的调用者处理。

然后是关于不可恢复的错误的处理。一是Sudden Death，保留足够的信息（如日志）后退出或重启模块；二是保证强异常安全的事务性回滚[7]。

回滚的实现主要有两种方式：事先复制可能被回滚的资源的副本，若回滚则使用副本代替已经发生错误的状态，否则丢弃副本；把回滚操作分解为若干具有无异常抛出保证的撤销操作[7]。后者比较节约资源，但受到限制比较大（需要对应的撤销操作都存在）。

若有必要，可以重新抛出与捕获的不同的异常（需要不同的异常类型时），或考虑捕获所有异常（在模块边界，不允许抛出异常时）。

其它情况下无需特定处理，保持异常中立。

VIII.异常规范

异常规范指定能从函数被抛出的异常类型，在函数声明（包括定义）中指定。C++11把旧有的异常规范称为动态异常规范，使用throw引导。违反动态异常规范会导致std::unexcepted的调用[3]。

动态异常规范有以下缺点[6][8-10]：

静态类型的不一致性——它不是类型的一部分（不能出现在typedef中），却限制函数指针赋值和虚函数函数覆盖；

运行时强制检查——不对程序员保证所有异常已被处理；阻碍编译器优化代码，影响性能；

无法有效地用于泛型代码。

在实践中，只有两种异常规范被认为是有用的：什么也不抛出，或者能抛出任何异常。

C++11提供了新关键字 `noexcept` 引导的新的异常规范，并废弃(deprecate)动态异常规范[3][10]，以改善这种状况。 `noexcept` 只表示是否能保证不抛出异常，而不限定具体的异常类型。违反 `noexcept` 异常规范会导致 `std::terminate` 的调用。

应该停用动态异常规范。适当使用 `noexcept` 异常规范，以使代码得到更多的优化机会。

IX.结论

异常是一项C++中的重要特性。使用异常需要注意异常安全，可以通过RAII惯用法实现。

错误可被作为一类异常对待。异常主要被应用于错误处理，并且一般优于其它方法。使用异常进行错误处理时应特别注意区分错误是否可恢复。

异常规范是C++提供的语言特性，用于检查可以抛出的异常的类型。现在应使用noexcept异常规范而不是过时的动态异常规范。

参考文献

- [1] Exception handling [G/OL]. 2013-03-09, 2013-03-10. http://en.wikipedia.org/wiki/Exception_handling
- [2] Double fault [G/OL]. 2012-05-02, 2013-03-10. http://en.wikipedia.org/wiki/Double_fault
- [3] ISO/IEC 14882:2011(E)，Information technology — Programming languages — C++ [S]．Geneva, Switzerland，2011.
- [4] Boost Community. Error and Exception Handling [A/OL]. http://www.boost.org/community/error_handling.html
- [5] David Abrahams. Exception Safety in Generic Components [C]. Generic

Programming, Proc. of a Dagstuhl Seminar, Lecture Notes on Computer Science. Volume. 1766. http://www.boost.org/community/exception_safety.html

- [6] Herb Sutter. Exception Safety and Exception Specifications: Are They Worth It? [J/OL]. Guru of the Week, #82 , 2001-06-30. http://www.gotw.ca/gotw/082.htm
- [7] 刘未鹏. 错误处理(Error-Handling)：为何、何时、如何(rev#2) [A/OL]. http://blog.csdn.net/pongba/article/details/1815742, 2007-10-08, 2013-03-10.
- [8] Herb Sutter. A Pragmatic Look at Exception Specifications [J]. C/C++ Users Journal, 20(7) , 2002-07. http://www.gotw.ca/gotw/082.htm
- [9] Herb Sutter. Questions About Exception Specifications [A/OL]. 2007-01-24, 2013-03-10. http://www.gotw.ca/gotw/082.htm
- [10] Doug Gregor. Deprecating Exception Specifications [J/OL]. C++ Standards Committee Papers, N3051=10-0041 , 2010-03-12. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3051.htm

《高质量C++/C编程指南》陷阱 2

Created @ 2013-04-12, r1 rev 2013-04-12, markdown @ 2015-09-14.

答 http://tieba.baidu.com/p/2262386913 。

本文提到的“原文”同样是指《高质量C++/C编程指南》。 原文链接已经修正。

(*)“以实际经验出发”并不和内容有误矛盾。即便实际经验是符合事实的，对于不够有经验的读者还是有相当的危害。

(**)文章中指出的错误大多是和公认的常识相悖，即便标准能够提供更精确的解释。即便没有标准，错误也不应该出现。

1

“作者的本意是想说明实际操作中通常的组织代码方式。”

↑虽然比较合理的可能的原意能够被有经验的用户通过下文猜出来，但是这个表述无疑是错的。即便不管标准定义，“程序”的概念在C++中是清晰的。声称一个程序通常组织为“两个文件”，是显然的误导。

1.2.1

这里需要补充，实际是否应该使用这种风格取决于具体需要。

类模版因为分离定义导致代码过于复杂所以往往不适用，但如果能保证可维护性也不是不能用（如 libstdc++ 拆分的 `.tcc` ）。

而除了模版外 `inline` 函数也适用。

甚至有实用的不用 `inline` 或非模版函数的 header-only 库（如 CImg ——虽然我内部依赖组织得不怎么样，嘛，题外话了）。

3.1

命名原则应该足够清晰。

需要指出，只强调“直观可以拼读”往往不是最佳实践。当一个“直观”标识符长度过长时就不能使用：

(1)实现不支持的情况。在ANSI C89只规定较少的标识符长度支持。在C++中这种情况的影响较少。

(2)大量出现，过长的标识符影响阅读时。一般拟定通用的缩写，在文档（至少注释）说明。

3.2

没记错的话 `namespace` 加入 C++ 是在 90 年代前期提出的，2001 年的主流编译器支持应该不成问题（ VC6 也支持）。这不是一个很容易实现错而需要避免使用的特性。

考虑到即便是很久以后不少用户对 `namespace` 的使用也有因为各种原因有意无意的回避反而导致放弃使代码更清晰潜力（例如 box2d 和旧版 FLTK ），在这里补充是有必要的。

4.3

已修正。

链接是BS的个人主页。失效似乎是最近几个月的事情。

5

由于历史原因，“常量”的用法比较混乱，具体含义往往需要通过上下文推测。这里对标题的解释不充分显著地增强了误导的可能性。

这里是非常基础的内容，不应该有放任明显混淆这样的低级错误。

“完全”当然不可能。是否“尽量”，也要看需求。例如有时需要用于条件编译的编译时确定的整数值，const在这里无能为力。

6

没找到在哪里说“帮助新手区分指针和引用”。出处？

帮助新手区分指针和引用完全不必要也不应该使用这种有问题的说法。

6.3.1

(1)的确需要补充。

(2)原文没有指出就是 `assert` 。 从契约式设计上来说，用 `assert` 是一种常见的实现，但使用在运行时抛出异常等其它手段有时也是可接受的手法。 而 C++11 补充的 `static_assert` （以及之前的模拟）也算，尽管适用范围受限。（题外话， `assert` 就没法 `constexpr` 了，这点大概算是 `C++11` 的缺陷。）

6.3.2

这点确实是错误。已补充修正。

不过需要另外指出，“临时变量”的说法是有问题的。C++中的变量是指声明引入的对象或引用，临时变量中的“变量”很多时候无法作为这个含义。从下文来看，是“临时对象”之误。

这样，就有一个硬伤。表达式中使用模拟类类型的对象初始化得到的一个非类类型右值，不是临时对象。

```
return int(x + y); // 注意：不创建临时对象。
```

至于后面“已经说过”的问题，重点是刻意区分“值”的意义。这又是一个混乱的概念，尽管严格意义上相对单一。不应该在此处引入增加误导可能。（我很怀疑原作者是否有意识到这点。）

7

和本文无关，不过提一下，根据 wikiquote ，这句不是 Bill Gates 说的。

7.1

这点是基础概念问题，和具体实现无关。 C 语言也需要讨论类似问题，虽然说法不同。

10.2

从 OOD 的角度来看是这样没有错，但注意文章强调的是编码而不是设计上的质量，说的是 OOP 。

从 OOD 到 OOP 的映射过程不是单一简单的，毕竟使用的不是建模语言。尤其是对 C++ 来说，不少语义和 OOD 所强调的抽象不相吻合，也不能方便地实现。

简化 OOD 到 OOP 的过程或许正是 Java 之流体现价值的重点之一。然而，这样的选择是以牺牲可能实现的灵活性为代价的。而这种灵活性正是 OOP 用户欠缺与需要重视之处。

这样的使用可以说是“一种技巧”，但类似地也有其它一些特性，如 class-scope `using` 。
C++ 为什么不抛弃这些看起来不符合 OO 方法学的特性？我认为主要并不是兼容性的问题，而是在设计上鼓励语言的用户有权利选择自由灵活的表达方式。 就这里的问题而言，`private` 继承的含义是“用……（基类）实现”，即便 OOP 角度上不是典型的组合，同样可以是 OOD 中组合的映射结果。

何勤《轻松学习C程序设计》简评

Created @ 2013-08-31, rev v1 2013-08-31, markdown @ 2015-09-15.

应作者要求，阅《轻松学习C程序设计——揭开计算机与程序设计的奥秘（修订版）》（资源通过网络找到），评论如下。

（此处找到的材料的节标号可能有错乱，按页面顺序评论。因此也不保证分节评论。）

1 计算机的重要性和基本工作原理

作者使用“理想厨房系统”作为计算机系统的比喻。有以下一些问题：

难以估计对于初学者来说这个比喻是否得当，有助于真正理解。

厨房中突兀地出现了“地址传送带”“控制传送带”“IR”“PC”“R0”等和喻体不直接相关的生造抽象。这些抽象是否真容易理解？后面虽然有比较详细的说明，但细节并不十分直观且篇幅不小，可能对读者造成负担。

语言风格问题。虽然总体比较清晰，但一些名词并不十分浅显易懂（相反容易觉得很“专业”——某种意义上事实的确如此）。如“取指周期”——具体什么意思，并没有在此直接解释。

《理想厨房系统与计算机系统术语对照表》中：

> 内存(又称为主存，包含很多大小相等的基本存储单元)

——错误。

虽然通常说的内存往往是指主存，但实际上是两回事。

在论述存储的体系结构时，习惯上“内存”指的是联机存储，和脱机存储即“外存”对立。

很容易找到反例，如智能手机等设备常见的 Flash ROM 属于内存，但它不是主存。这些设备的主存使用 RAM 。

此外，“内存”作为英文 memory 一词的翻译，也可以指一般的存储，没有特别的“内”的意思。只不过通常在 PC 等设备上程序必要的主要存储就是联机 RAM 主存抽象得到的，主存、内存和 RAM 三个概念有时候就会被混用。（后文有区分 RAM 和内存，但并没有完全论述清楚。）

> 一个字节（即8位）

——一个字节即 8 位也只是通常情况，不总是成立。

下文

> 字节：一个8位的二进制的位串，就构成了一个字节（Byte）

——在这个意义上显然是错误的。

正式地说，在体系结构中的字节和八个位组成的八元组/八位组(octet) 是两个相对独立的概念。只是通常一个字节是 8 位容易导致一些错觉。

习惯上，作为存储容量的计量单位，一个字节被默认为 8 位组成， ISO/IEC 80000-13 规范化了字节的这种含义。但是，更严格的场合中，如通信、编码等规范化时，会严格区分两个概念。

特别地， C 语言明确允许一个字节大于 8 位。考虑到 C 语言是本书讨论的重点，在这里混淆这两个概念是不合适的。

二进制数介绍比较详尽，但较复杂，初学者可能比较难接受。

> 操作系统又被称为系统软件

——系统软件是个比较笼统的概念，但显然不止包括操作系统。

> （称为可执行程序，程序文件名的扩展名为.exe或.com）

——显然错误。文件扩展名只是特定环境（如操作系统）下的约定。文件是否表示可执行的程序，取决于文件的内容以及环境是否允许。

本章内容比较多，可能缺乏一些典型的简单抽象，如冯·诺依曼结构只是一笔带过。对于读者而言，一口气理解大量术语和比喻（是否一定有助于理解还有疑问），很可能比记忆经典的简单抽象更困难，且更难以保证理解内容的准确性。

> 不过，Lisp，Forth这些函数式语言与以上所列这些命令型或面向对象型语言有较大的区别

—— Forth 不是典型的函数式语言，相反，它的主要范型是指令式/命令式(imperative) 的。

2 C语言的基本概念

2.1.2 C语言程序的一级构成成分——函数

“一级构成”是无稽之谈。

> 参照本书末附录A的上机指导，在Windows操作系统环境下，使用VC++6.0编译并运行以下C语言程序（虽然这是一个C++语言的编译器，但该编译器也很适用于对C语言的源程序进行编译和调试。只不过要记住：C源程序的文件名一定要以“. c” 作为扩展名。注意：此处扩展名用的“c”是小写，不能用大写。如果你忘记了加上这个扩展名，该编译器就会将你的程序当作C++的源程序进行编译。

在扩展名上又一次的低级错误。注意 Windows 原生的文件系统（ FAT 和 NTFS 等）上虽然可记录文件名中字母的大小写，但是把仅有大小写差异的文件名视为相同文件的文件名。所以 VC++ 区分大小写是完全没有必要的。

通常类 UNIX 环境的文件系统严格区分文件名中字母的大小写，使用的编译器会 .C 作为 C++ 程序的扩展名。

> 一个C语言源程序类似于某个特殊的公司，main()函数的角色类似于公司的总经理（该公司的特殊性在与：每个员工所负责的工作都是互不相同的）

——尽管下文有表指出大致的对应关系，但缺乏直观性，略牵强。

2.2 C语言程序的二级构成成分——定义、语句、注释和预处理命令

标题即体现抽象层次混乱。事实上这四个概念中没有任何两个是并列的。而且，从严格规定的翻译阶段(phase of translation) 来看，注释和预处理指令比上文所谓的“一级”成分函数更高。因此这是显然错误的。

> 其中的PI是一个符号常量

——符号常量是个现时不常用的老旧说法，不是 C 的正式概念，且容易引起混淆。（什么是“常量”？）

> 这是一条本章将要重点讲解的赋值语句。

—— C 并没有单独的赋值语句。

使用了 C99 引入的单行注释“//”，但和本书其它部分强调 C89 和 C99 的区别不同，这里没有提到 C99 。（虽然 VC++ 到 2013 年都没完全支持 C99 ，不过这个倒是有扩展支持了。）

此外，关于标准的版本，正确说法是 ANSI C89 的正文被接受为 ISO C90 ，而 ISO C99 在 2000 年被接受为 ANSI C 标准。之后标准一般直接称为 ISO C 。本书所谓的 ANSI C99 的提法不合适。

> C语言程序的二级主要构成成分，分为两大类：定义序列和语句序列。在函数体中，定义序列在前，语句序列在后

——显然错误，并且无视了声明的重要性。

> 高级语言源程序中的定义，是用来定义变量和定义（用户自定义的）类型的

——漏了函数，非常不妥。下面

> 每一条定义都要以分号结束

显然也是错的——考虑函数定义。

> 高级语言源程序中的语句，其实就是用来告诉编译程序：我们想要计算机对于源程序中以变量或常量形式出现的数据，执行什么样的运算（算术运算还是逻辑运算等等）和操作（取数、存数、输入、输出）；或者，我们想要计算机根据哪个表达式的计算结果，去选择下一条要执行的语句（这句话，你或许要学了选择结构这一章以后，才能真正懂得）。

这一段再次暴露了抽象的混乱。实际上，在 C 这个层次上，运算完全可以归类为操作的一种，并不需要区分存取和算术操作——而这些操作的语义蕴含于表达式（不是语句）的求值中。

而 ISO C 使用的方法是定义一个抽象机，操作表现为这个抽象机的行为。存取可能是表达式求值包含的副作用（具体地，对外部环境的 I/O 操作和向对象存储值一定是副作用，volatile 左值的读取也是副作用）。

> C语言源程序的二级次要构成成分是：注释、编译预处理命令和声明（只是对在别处出现的定义，起着辅助说明作用。请参见函数一章）

——错误。直接无视了声明作为定义，相当于篡改了“声明”的概念。

> 命令型（或称为过程型）高级语言源程序的本质

——命令型和过程型不是一回事。关于这点在《面向对象和所谓的“面向过程”》已经有过澄清。

> 与大多数其他高级语言不一样，在编译C源程序之前，都必须事先运行一个编译预处理程序

——错误，不在所有情况下适用。尽管完整的翻译阶段包含预处理，但是预处理后的中间程序可不需要再次预处理，且仍然可以是 C 源程序。

> 对源程序进行一些（通常是少量的）辅助性的插入、替换和编辑工作。

——实际情况通常很不“少量”。可以观察 GCC 等常用实现的预处理后的程序，比较一下和原始代码的大小。

> 但也有一些编译预处理命令——比如条件编译命令——是可以书写在函数体内部的

——这里的说法比较含糊。预处理本身不禁止指令和函数的顺序，事实上通常预处理器的实现无视函数。尽管不是常规做法， `#include` 等指令写在函数体内部也可行。

2.1.6 C语言源程序的编写、编译、链接和调试过程（参见附录A）

> 这里的流程和 VC++6 实际所做的不一样。例如， `nmake` 的调用被无视了。虽然可以理解这样描述是为了方便新手阅读，但接下来关于编译器的行为也是很含糊的。

之前本书有提到“编译程序(又称为编译器)”——实际上这个说法是不严格的。典型的实现中，供用户通过命令行接口调用有若干个程序（可执行程序或库），它们可以总称为编译器。其中一个程序称为编译器驱动程序(compiler driver)，负责调用其它程序完成预处理等，有时候也被称为编译器。

对于 VC++ 来说， C/C++ 编译器的驱动程序一般是 cl.exe 。 VC++ 中 cl 调用了附带的动态链接库完成翻译的不同阶段，一般笼统地称为编译——而链接则是 link.exe 完成的。

对于 GCC 来说，由于按 *NIX 传统倾向于直接调用分别调用各个静态链接的可执行程序而不是动态库，这点更加明显：可以很容易观察到占用最多时间的往往是实际执行代码生成工作的程序（例如 MinGW 和 Cygwin 下的可执行文件名是 cc1plus.exe ）。

> 不走弯路，通过一本书就能真正掌握编程的基本思路和技巧，就是最短最快的捷径

——这个有 Peter Norvig 的吐槽大概就够了。

2.3 C语言源程序的正文部分

所谓的“正文部分”仍然是生造的概念。

2.4 C语言的字符集

> C语言源程序的全部正文部分，都只能够使用如下所列举的字符来构成

——其它地方说的好好的 C99 呢？

此外，标题党。这里所说的实际上是基本源字符集(basic source character set) 。明明标题是“ C 语言的字符集”，另一个重要的基本执行字符集(basic execution character set) 却一点不提。

全角和半角也有乱掉的情况（下略）。

2.5 标识符

> 在高级程序设计语言中，我们通常用标识符来命名，我们想要用计算机进行加工的其数值可以变化的数据——变量（→）、不可变化的一部分数据——符号常量

——又来了个“符号常量”的混沌说法。

2.6 关键字

> “关键字”有的教科书又称为“保留字”

——这里补充一下，对于 C 来说问题不大，但有些语言后者范围更大：如 Java 的 `goto` 是保留字但不是有意义的关键字；再如 C++ 的 `and` 等 alternative token， 尽管不是正式说法，也可被称为保留字。

> 切记：不要将关键字作为普通的标识符来定义和使用

意思容易理解，但是说法不严谨（考虑到后面提到了“分隔符”之类的词法问题，这里的漏洞更加严重）。

在 C 语言中，术语“标识符(identifier) ”出现在两个地方：一种是作为预处理记号(preprocessing-token)， 此处只有标识符没有关键字；另一种是记号(token)， 其中包括关键字和标识符等。

即预处理之前的标识符被预处理后分化为关键字、标识符和其它记号。

2.7 分隔符

标题的分类是胡扯。无论是 seperator 还是 delimiter 都不像。

> C语言字符集中的空格，逗号，回车/换行（ASCII码为13）这三个字符在源程序中起着分隔的作用

——词法和语法混起来讲先不论，制表符的存在感呢？

> 在同类项之间作分隔：要用逗号，空格则可加可不加

——什么叫“同类项”？后面有例子，但是比较笼统——似乎只是说了变量名之间算同类项，但又无法让读者知道是不是有其它适用的外延。

2.8 常量

> “A. 整型常量 567，-425， 0 等， 是没有小数分量的数值”

——整型常量？是指整数常量(integer-constant) ？负号是什么情况（看来又是谭×流么）？

> B. 实型（浮点型）常量

——“实型”看来又是谭×流的说法。

浮点数十六进制表示和二进制阶码果然没存在感。

> 而是通过采用编译预处理命令中的宏定义的符号常量（→）来处理此事。即用标识符来命名的常量

——完全胡扯。常量(constant) 的语法里根本就没有这号玩意儿。

2.9 变量

> 比如register int num; 编译器就很有可能将变量num的存储单元安排在CPU的寄存器中）

——现代的编译器比较少理会 register 关键字，因为往往编译器比用户更清楚到底是不是存进寄存器里比较高效（ C 还好点， C++11 直接 deprecated 掉了）。

> 给要使用的变量起名字，正式的术语称为定义变量

——胡扯。“ `extern int i;` ”显然可以不是定义，但它就是取了个名字。

> 对于简单类型的变量

——没说清什么是“简单类型”，大概是想说声明符可以放在要声明的标识符一侧，不用顾及函数或数组的声明符。当然 C 在这里是比较坑。

> 变量名一般要用标识符来命名

——难道还可以不用标识符命名？

> 任何一个int 型的整型变量，在VisualC++ 6.0编译环境下都被编译程序分配了4个字节的内存空间作为存储单元；在Turbo C 2.0编译环境下被分配了2个字节的内存空间作为存储单元。

——这种事无巨细的说法是很糊弄的风格。凭什么就不能是 8 个字节？再者，如果整个优化掉了呢？

> 在程序运行中，以实数值形式（即有小数分量）出现的变化着的数值（比如34.1，-678.34等）

——无理数数算不算有小数分量的实数？

> 单精度浮点型变量的有效位数是十进制的7位

——并非刚好 7 位。

> 任何字符型的变量，在内存空间都被编译程序分配了一个字节的内存存储单元，用来存放该变量所对应字符的ASCII码（大多其他高级语言也都是这样）

——又是常见错误。

上面提到过了， C 有基本执行字符集。这决定了运行时存储的整数和所表示的字符的关联。

更重要的一点是，无论是基本源字符集还是基本执行字符集，都只是说要至少包含哪些字符，没明确具体实现为什么字符集什么编码。

ASCII 只是最常见的实现。一个不兼容的 ASCII 的例子是 EBCDIC 。

[高级点的技术交流]什么叫语法(syntax)

Created @ 2015-07-01 08:50.

先声明，我不是撸这专业的；然而大都数吹逼材料在这个问题上的犯傻过于普遍，都不到点上，太不像样了，以至于熟练工还真能一本正经口胡……

原因大概是从头就错惯了，以讹传讹（就像什么“变量”“堆栈”的意思一样）。

1.词义

中文维基被墙所以照搬英文维基解释翻译一下。

词条 syntax ：

> In linguistics, syntax is the set of rules, principles, and processes that govern the structure of sentences in a given language. The term syntax is also used to refer to the study of such principles and processes. In linguistics, syntax is the set of rules, principles, and processes that govern the structure of sentences in a given language. The term syntax is also used to refer to the study of such principles and processes. The goal of many syntacticians is to discover the syntactic rules common to all languages.

在语言学中，语法是一个语言中控制句子结构中的规则、原理和过程。术语 syntax 也指对这些原理和过程的研究。（译注： syntax = 语法学。）许多语法学家的目标是发现适用于所有语言的语法规则。

> In mathematics, syntax refers to the rules governing the behavior of mathematical systems, such as formal languages used in logic. (See logical syntax.)

在数学中，语法指控制数学系统行为的规则，例如逻辑学使用的形式语言。（参见逻辑语法。）

> # Etymology
>
> From Ancient Greek: σύνταξις "coordination" from σύν syn, "together," and τάξις táxis, "an ordering".

语源

古希腊文 σύνταξις “协调”，由词素 σύν(syn) “在一起” 和 τάξις táxis “一种秩序”组成。

语法学的历史非常悠久，公元前 4 世纪古印度语法学家波你尼(Pāṇini) 撰写的梵语语法《八章书》(Aṣṭādhyāyī) 是一个重要的代表……余略……

事实上，抛开自然语言的历史，词条 syntax (logic) 给出了更贴近这里要讨论的上下文的定义：

> In logic, syntax is anything having to do with formal languages or formal systems without regard to any interpretation or meaning given to them. Syntax is concerned with the rules used for constructing, or transforming the symbols and words of a language, as contrasted with the semantics of a language which is concerned with its meaning.

在逻辑学中，语法是任何形式语言或形式系统中不考虑任何解释或含义的部分。语法和用于构造或转换一个语言中的符号和词的规则有关，这和语言的语义——和含义相关——相反。

> The symbols, formulas, systems, theorems, proofs, and interpretations expressed in formal languages are syntactic entities whose properties may be studied without regard to any meaning they may be given, and, in fact, need not be given any.

在形式语言中表达的符号、公式、系统、定理、证明和解释是语法上的实体，它们的属性可以在不考虑解释和含义的情况下被研究——事实上也不需要。

> Syntax is usually associated with the rules (or grammar) governing the composition of texts in a formal language that constitute the well-formed formulas of a formal system.

语法通常和控制形式语言中决定合式公式的文本的构造的规则（或文法）关联。

> In computer science, the term syntax refers to the rules governing the composition of meaningful texts in a formal language, such as a programming language, that is, those texts for which it makes sense to define the semantics or meaning, or otherwise provide an interpretation.

在计算机科学中，术语“语法”指在形式语言中控制构造有意义的文本的规则，例如对一个程序语言，即明确对定义语义或含义有意义或提供一个解释的文本。

关于形式语言的语法学，实际上是现代数学基础最重要的分支之一（数理逻辑）的主要组成部分。对此的延伸话题限于篇幅不作展开，有兴趣的读者可以查找词条内的链接和参考文献进一步了解术语之间的联系。

2.文法(grammar)

细心的读者可能会发现，实际上经常被称为语法的词语并不是指 syntax ，而是上面出现过的另一个容易混淆的词： grammar 。

仅涉及自然语言的话题中， grammar 常被翻译成“语法”，而 syntax 则是“句法”（上面的 logical syntax更惯于被翻译成“逻辑句法”），这并没有造成太大问题。不过在涉及更广的范围下，这样会造成一些误会。因为“句”的概念的微妙差异，导致“句法”这个词失去基础。这个情况下，作为泛指，通译 grammar 为“文法”， syntax 为“语法”。

按维基给的定义：

> In linguistics, grammar is the set of structural rules governing the composition of clauses, phrases, and words in any given natural language. The term refers also to the study of such rules, and this field includes morphology, syntax, and phonology, often complemented by phonetics, semantics, and pragmatics.

在语言学中，文法是指任意一个自然语言中控制组合分句、词组和词结构性规则的集合。这个术语（译注：语法学）也指对这些规则的研究，这个领域包括（词语）形态学、语法学和音韵学（译注：不要和后面讲的 phonetics 混淆），经常也涉及语音学、语义学和语用学。

可见，至少在自然语言中，文法学的研究范围涵盖了语法学；作为研究对象，语法现象是文法现象的真子集。

对于形式语言来说，没有系统的音韵学或语音学研究的必要。但是，其它重要的部分，尤其是语义和语用这样明确不属于语法学研究范围的内容，仍然是重要的文法现象。

因此，随意混淆“语法”和“文法”是有问题的。在文法的范围内偷换“语法”是典型的缩小外延的逻辑谬误。

3.语义(semantics)

上面的讨论实际上还省略了一个重要的问题：什么是“含义”(meaning)，和“语义”又有哪些差异？

这里一并澄清关键点：前者泛指所要表达的内容即“目的”，而后者是指对含义的一般研究，尤其侧重和被使用的语言的联系。

提这个问题是因为设计和使用人工语言需要注意的一些现象。人工语言的语法并非自然演化而是人为设计的，仅从动机来讲语法背后往往存在非常具体的含义。但是，这些语言的语义和设计这些语言的语法时已经被考虑并集成在语法规则内部的含义并没有关系。

这样的一个明显好处是，语义规则可以完全和语法规则分离讨论（不像自然语言反复折腾上千年也不会有完全的标准形式）；对于用机器实现语言，语法和语义规则的检查（前者又称为分析(parsing)）也能够容易被分离，便于复用。

因为缺乏设计和实现语言的机会和必要训练，这个和自然语言不同的特性往往造成不少初学者几乎永远都没有机会搞清楚问题，而对解决一些问题（例如学习新的程序设计语言）造成一些阻碍。

这里的设计要点是有普遍实用意义的。形式文法可以同时描述语法规则和语义规则，但相比之下不管是理论还是实践前者远远更加成熟。具有文法表达的形式语义通常是指称语义或公理语义，这对于大多数语言设计者来说要求过高，且现阶段因为各种局限性仍然缺乏实用价值；所以大多数语言规范并不会给出形式文法指定的形式语义。若在设计时不对语义和非语义部分加以区分，则实现语言更加混乱和困难。

与形式语义的困难相反，基本上任何现在的人工语言都会给出形式语法——一般通过 BNF 范式之类的领域特定元语言来描述。这种形式描述的主要优势是能够对处理语法的系统（主要就是语法分析器）提供自动化实现，例如分析器生成器（经典的如 yacc ）或分析器组合器（如 Parsec ）。因为可以节约大量编写和维护分析器的工作，这些自动化工具为调整语言的语法设计带来了极大的便利。（当然，并不是所有语言都方便这样做，下面的例子会提到。）

由于形式语法的高度普及，有时候提到syntax就指得是形式语法，而其它构造规则可能并不称为语法——尽管它们部分或全部地是传统语法学研究范围的内容。

4.词法(lexical syntax/lexical grammar)

在语法中，因为有些最基本的和字符序列直接相关的部分内容的含义相对固定，而不需要特别指定不同的规则，习惯上把这部分语法单独提出称为词法。

在实现上，词法分析和其它更一般的语法分析也可以分离并单独配置；同时，也便于学习者理解语言的语法构造。实践证明这是自然的，因此这种惯例普遍存在。

这样，也可以提升词法到和其它一般语法并列，直接使用文法描述。

当然也不是所有语言都必要单独提出词法（除了指定允许什么字符以外），比如brainf*ck或者组合子逻辑以及这样的语法上异常“简单”的语言。

5.实例

在 ISO C Clause 6 中，语言的形式语法在标题 **Syntax** 下澄清，而剩余部分澄清非形式的翻译时检查的约束条件 **Constraints** 和一般的语义规则 **Semantics** （对于库还有 **Runtime Constraints** ）。

ISO C 区分 **Constraints** 的要点是——这些规则和 **Syntax** 一样，必须在翻译时被检查和诊断(diagnostics)。

WG14/N1570

> # 4 Conformance
>
> 2 If a ‘‘shall’’ or ‘‘shall not’’ requirement that appears outside of a constraint or runtimeconstraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words ‘‘undefined behavior’’ or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe ‘‘behavior that is undefined’’.

这个意义下， **Constraints** 的规则和语法规则具有同等效力，尽管其中有一部分是形式语法说不清楚的(*constant-expression*) 。而剩下的，违反了就当未定义行为了。

在ISO C++中，这里的设计更加明确。

WG21/N4527

> ## 1.3.27 [defns.well.formed]
>
> **well-formed program**
>
> C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).
>
> # 1.4 Implementation compliance [intro.compliance]
>
> 1 The set of diagnosable rules consists of all syntactic and semantic rules in this International Standard except for those rules containing an explicit notation that “no diagnostic is required” or which are described as resulting in “undefined behavior.”

同逻辑学的习惯，“合式”被作为构造性正确的标准，是可以明确决定的——只不过仍然不要求实现必须确定。有别于未定义行为，这里明确区分了诊断要求，表示了更精细的区别。

ODR 本质上来源于构造性语义规则，不过因为单独翻译链接模型和对于不同实体本身的复杂性，这里做了妥协，直接和语法规则、可诊断语义规则并列了。

另外值得一提的是，虽然 ISO C 和 ISO C++ 的 Annex A 都直接给出了类似的东西，但标题却不大一样： C 是 **Language syntax summary** ， C++ 是 **Grammar summary** 。

其实也不难理解： C++ 的语法和语义虽然在原则上多少是想要明确分离的，具体设计上却自己打脸。

C 的语法接近上下文无关语法(context-free grammar) （考虑到抽风的指针语法所以不纯粹），而C++的情况明显糟糕得多。

因为语言要求的一些消歧义非形式文法规则，严格的、单纯的 C++ 语法分析器事实上并不存在。比如说声明中的 `()` 表示一个函数声明符还是初值符的一部分，不判断现有实体的类型是不知道的，而类型需要经过进一步上下文相关的语义分析才能确定。现有的实现基本上用的是有效然而无奈的笨办法：假定每种文法正确都试一遍，最后留下一个没错的就说明没歧义了。

换句话说，要说 C++ 的“语法”烂，正是因为它连“语法”是什么都说不清楚——规则的边界本来就是模糊的。

大概这个槽点， ISO C++ 的 Annex A 也就老实写 grammar 而不是 syntax 了，尽管和C一样，明明两者都没有给出形式语义……

Java 在这里就好得多，直接给出了文法定义，并明确使用上下文无关文法。

（尽管实际上仍然只有形式语法……是 Guy Steele 参与起草的关系么？）

JLS 8

> 2 Grammars 9
>
> 2.1 Context-Free Grammars 9
>
> 2.2 The Lexical Grammar 9
>
> 2.3 The Syntactic Grammar 10
>
> 2.4 Grammar Notation 10

当然， Java 的语义一锅乱七八糟的这里不展开了。

C# 大体上抄的 Java ，以及C++的部分糟烂（又是关于<>的疼货）……

ECMA-334 3 2005

> 9.1 Programs
>
> 9.2 Grammars
>
> 9.2.1 Lexical grammar
>
> 9.2.2 Syntactic grammar
>
> 9.2.3 Grammar ambiguities

ECMAScript 总体上也是一路货（否则干嘛叫 JavaScript ……）

ECMA-262 5.1 2011

> 5 Notational Conventions
>
> 5.1 Syntactic and Lexical Grammars
>
> 5.1.1 Context-Free Grammars
>
> 5.1.2 The Lexical and RegExp Grammars
>
> 5.1.3 The Numeric String Grammar
>
> 5.1.4 The Syntactic Grammar
>
> 5.1.5 The JSON Grammar
>
> 5.1.6 Grammar Notation

C++/CLI 是直接照搬 ISO C++ 偷懒了。

Haskell 2010 Report 则只是提了下 notation， 看起来都没说啥 grammar ……不知道对混乱的设计有多少自知之明呢。

作为补充说明， Scheme 的 R^{n}RS 是少数给出了文法描述的形式语义的语言规范。正文符号太疼略。

顺便，注意 Scheme 的语法和 S-expression 的语法是两回事。

6.结语

抛开形式语法，没有一个实用语言的语法是足够简单到直接能一看就全记住的……

那些提啥“只是语法”还有“语法书”的，真拿得出一本出来么。或者替我贴完 ISO C 的 Annex A 科普一下瞧瞧？

（剩下的懒得写了自行脑补。）

特性和设计

C & C++广义类型系统缺陷

Created @ 2013-04-16, r2 rev 2013-04-16, markdown @ 2015-09-15.

类型(type) 和 `const` / `volatile` 限定符(qualifier)

类型是程序设计语言中广泛使用的重要特性，被用于抽象数据和程序行为。类型之间遵循语言的若干语义规则，这些规则和类型一起被总称为语言的类型系统(type system) 。

C 语言具备较完整的静态类型系统，类型能够通过组合产生新的类型（数组类型、函数类型、指针类型等）。 C 的类型对象类型（包括作为不完整对象类型的 `void` ）和函数类型。 C++ 在这个层次上没有太大改进，增加了引用类型，而把void类型排除在对象类型之外。

`const` / `volatile`限定符通过限定对象类型体现主要语义（但也用于 `void` ），是类型的组成部分。

关键字 `const` 在 20 世纪 80 年代引入 C 语言，表示禁止写入对象的只读语义，以便实现利用只读存储器[C++ D&E]。标准化 ANSI C 时使用了类似的语法和语义规则增设了 `volatile` 限定符，语义为读取被限定的对象存储值和写入值时也产生副作用，禁止实现缓存读取的值而减少读操作，能有效提供对I/O寄存器等的映射[C99 Rationale]。

而 C++ 中的 `const` 相对有只读以外更激进的语义：对于特定的类型构建（能在编译时确定的）常量表达式。这在某种意义上提升了优化的可能性，但造成了语义的复杂。（题外话， `volatile` 在 Java/C# 有其它不同的含义。）

左值性(lvalueness) /值类别(value category)

ISO C 以及 ISO C++98 和 ISO C++03 的左值性(lvalueness) 是一种二值系统：C的取值分为左值(lvalue)和非左值；C++中分为左值和右值(rvalue) 。任何表达式的左值性取值都是二者之一[ISO/IEC 14882:1998, ISO/IEC 14882:2003]。

ISO C++11 的值类别(value category) 是左值性更复杂的演进，基本类别构成三值系统：纯右值（ prvalue ，相当于 C++03 的右值）、消亡值（ xvalue ，新增的值类别）和左值(lvalue) 。其中纯右值和消亡值合称右值，消亡值和左值合称泛左值(glvalue) 。任何表达式的值类别取值都是三种基本类别之一[ISO/IEC 14882:2011]。该演进主要是因为右值引用的引入而显得必要，详细动机参见[ISO/IEC JTC1/SC22/WG21 N3055]。

左值性来源于 B 语言需要对表达式的左操作数的做出语义限定的需要：只有左值(left-value) 才能作为“=”、“++”和“--”的左边（……以及“&”的右边）——此时 lvalue 显式地作为文法元素；类似地，出现在“=”右边的元素相对称为 rvalue[User's reference to B]。左值的概念后来被沿用至 C 和 C++ 语言。

由于左值也同时被作为 `&` 操作符的操作数的语义限定条件，实际上表示可能在内存中保持的对象——存在对用户可知的对应的内存位置(memory location)， lvalue 也被叫做 locator value ；而C中所谓的右值(rvalue) 一般即为值(value) 的同义词[ISO/IEC 9899]（ C++ 的值则可以是实现定义的其它概念）。

注意 C 和 C++ 的左值性在函数类型的表达式上有差异： C 的函数指示符(function designator) 不是左值也不是右值。而 C++ 的函数名是左值。

左值变换(lvalue transformation)

以上讨论的是原始的左值性/值类别。在 C/C++ 中，通过特定的、仅有少数求值（语法）上下文例外的隐式转换，转换的结果的表达式的左值性/值类别可以改变，同时类型也可以改变。

这些转换在 C++ 中为便于重载解析(overloading resolution) 描述而被统称为左值变换，包括三种标准转换(standard conversion) ：左值-右值转换(lvalue-to-rvalue conversion) 、数组-指针转换(array-to-pointer conversion) 和函数-指针转换(function-to-pointer conversion) ，其中值类别由泛左值转换为纯右值（对应 C++98/03 中左值转换为右值），同时后两者有类型变换，分别为数组左值 e→&e[0] 和函数左值 e→&e 。

转换被排除的少量上下文是作为（按 C++11 的说法）非求值操作数(unevaluated operand) 时、操作数需要左值时以及初始化左值引用时。

在C中也存在类似转换，排除的上下文也类似（当然没有用于初始化引用的情形）， 但只有第一种被明确命名为左值转换(lvalue conversion)[ISO/IEC 9899]。

左值性/值类别虽然不是 C/C++ 的名义上的类型，但从目的（静态分析区分操作、转换以及检查类型错误）来说，符合静态类型系统的一般特征。因此本文称为广义类型一致处理。

左值、限定符和引用类型：冗余和复杂

一个主要的问题是左值性和 `const` / `volatile` 限定符（作为类型的组成部分）的不完全正交组合造成的冗余。

对于 C 以及 C++ 中排除类类型(class type) 外的非左值表达式， `const` / `volatile` 限定符被自动丢弃。这样，存在5种等价类：

- 非左值：不可写、读不产生副作用；

- 左值：可写、读不产生副作用；
- `const` 左值：不可写、读不产生副作用；
- `volatile` 左值：可写、读产生副作用；
- `const volatile` 左值：不可写、读产生副作用。

显然，这里非左值和 `const` 左值在写权限和副作用是重复的，除了作为 `&` 和若干转换的操作数。

对于 C 来说，这样的区分尚且是有意义的：`&` 能返回 `const` 左值的指针来间接获得所需的值，代价是需要占用存储；而非左值则只能立即使用值，但不需要被储存。

而在 C++ 中，这样的设计就有些匪夷所思了：明明存在能绑定右值的 `const` 左值引用来获取右值对应的值，为什么还需要 `const` 左值？

对于 C++ ，另一个冗余是，引入（左值）引用表示的左值，除了存储以及参与不同的重载（但显然能和对应对象类型产生歧义）外和对象左值没有区别，这对于内建表达式基本没有意义，保留不相同的规则造成不一致性。

事实上， C++ 中，（左值）引用正在取代传统非引用类型的左值的地位，早在 1992 年标准化开始时的第一篇公开论文[X3J16/92-0053 WG21/N0130]就提议这点，并在之后各种场合改头换面、缺乏直觉规律地出现（如模版左值类型推导、 `decltype` ）。但内建操作符表达式左值的类型并没有修改（比如内建赋值和前置返回引用）， 不和其它 C++ 特性一致而和 C 保持一致。这点很容易被忽视而导致一些误解。

可修改(modifiable) 左值的意义

在 ISO C/C++ 中，直觉明确的可写权限并没有被定义，而是规定了对左值的可修改(modifiable) 的概念，事实上表示可写左值的子集。

这个概念的一个重要应用是禁止数组左值作为赋值的左操作数：ISO C中数组类型的左值就明确不是可修改的。但事实上并没有明确违反语义规则时究竟有没有已经左值转换（转换为右值不能在“=”左边——如果这点成立，可修改左值的规则在这里是冗余了）。这种含糊的归类导致标准在起草措辞(wording) 上的一些问题，例如在 ISO C++ 中遗漏了明确的数组左值的可修改性表述（虽然可以推测合理的情形是和 ISO C 一致）， 可能是潜在的缺陷。

另外，对于位域(bit-field) 类型，可能是因为按字节地址寻址的困难性，尽管仍然能作为可修改的左值，也需要额外明确。可修改在语义上无法帮助减少这里的措辞。

为什么 C/C++ 在这里不够理想：解决的困难

抛开现有的语言特性，从设计新语言的角度看，相关的基本需求不难总结：

作为基本抽象，对象或者值需要可读；

不同于所谓“纯”函数式语言，能够保存状态，允许（但不滥用）副作用带来的表达计算的便利是 Algol-like 语言的基本需求之一；

作为能够容易操作硬件的语言，需要类似使用 `volatile` 的机制提供映射硬件存储的能力。

下面可以证明满足以上需求的设计可以比现有的更简单，而不必要有如此多的冗余和模糊（注意，适合实现或者通过现有语言演进，是另外一回事）。

显然，不完全正交的广义类型不如合并为有以下分类的一个类型系统显得更清晰： 值：可写、读不产生副作用；

- `const` 值：不可写、读不产生副作用；
- `volatile` 值：可写、读产生副作用；
- `const volatile` 值：不可写、读产生副作用。

这种方案要求需要对所有值一视同仁地提供 `&` 等现在的左值具有的操作，若不考虑彻底放弃或加上使问题更复杂的其它限制的话。（生存期需要另外一些规则，但相比之下不是太大的问题。）表面上看，这阻碍了值“不要求存储”的优化；但是除了 `volatile` 类型外，关于抽象机的可观察行为的等价语义可以很容易挽回这一点，至少对于 C 来说最终只有一点关键的阻碍—— `&` 的结果是什么？

基于现在的 ISO C/C++ ，至少理论上并非不可解决：内建&的结果是一个指针类型的非左值，一个重要的语义限制是表示“地址”——但地址相关的可能的可观察语义（指针算术以及值的标准输出）是实现定义的。尽管改变现有的、典型的把地址的取值范围映射至某个地址空间的实现附加的实现复杂性会是非常致命的。

但是更致命的是，关于“地址”和“内存位置”等存储实现的（反）抽象：按特定基本单位（字节）连续的存储。这种过于具体的约定换来符合现有体系结构的可实现性（以及关于“布局”的控制，如 `offsetof` 的易实现性），却为类型系统的简化制造了麻烦，同时削弱了用户对象类型的控制：无法抽象出不确定布局但占用存储的类型，也没有能力让用户直接使用标准的手法简洁一致地安排布局。（非常讽刺的是， C++11 在内存模型的定义上更严谨更全面，严格意义上的限制却也更大了。）

C++ 的问题更严重些。一个问题是引用类型初始化的非对象复制初始化语义。这点其实也可以通过规定激进的非 as-if （可观察行为）语义来解决（现在ISO C++关于特定的复制/转移构造就这样做，甚至可以无视副作用），虽然这样总体上的语义或许会更模糊，让用户更无所适从。另一点是，在内存模型和布局上一直无法完全由用户控制（如对于有虚函数的类，通过 `offsetof` 计算数据成员地址就无法保证结果可预期），这同时具有贴近现有体系结构实现和便于高层抽象的优点，但另一方面也可以说都是缺点。　　最后还有一个非常现实的困难：兼容性。假设 C 或者 C++ 确实能通过整体改换设计（幅度显然比当年 C 到 C++ 大得多）解决了这样的问题，新的语言规范如何适应旧的程序？如何使用户能够顺利迁移？从现状（ C/C++ 的用户，以及现有需要维护的项目）来看，这是明显不实际的。

结论

基于以上讨论可知，要消除类型系统的冗余带来的负面影响涉及过多的核心设计，从现有语言为起点立即改进这些缺陷可以认为是不可行的。对于用户而言，尽量清晰掌握核心概念完成任务是比较实际的。但对于语言自身的维护而言，这个问题导致加入新的语言特性、使不同特性有效协作的难度大大增加，需要长期努力才有望缓解。

转移 vs 复制

Created @ 2013-06-19, r1 rev 2013-06-19, markdown @ 2015-09-15.

ISO C++11 引入右值引用以及转移构造等机制作为转移语义的内建支持。对应原本的复制构造等机制抽象的复制语义，可以说和转移至少在语言层次上是并列的。　　那么转移和复制有什么共同点和区别？它们的关系如何？以下讨论几个相关的问题。

1.不论具体的语言支持，一般意义上的复制/转移语义抽象

复制和转移是使程序从一个执行状态迁移到另一个执行状态的操作。不过依赖的概念可能有些模糊。方便起见，这里约定“值”表示某个抽象的单一（可分辨）状态，假设“变量”是可变的值的容器。

对于最简单情况，值可以取空值（记作 `e` ）和非空的非平凡值（记作 `v` ）。

（不过，一般这样的抽象是不够的，因为复制和转移本质上表示的不仅是决定性(deterministic)操作，前后两个状态可以有未指定的值。因此可以引入一个表示变量的未指定的取值 `?` 。）

设程序 `P` 的两个执行状态 `E1` 和 `E2` ，每个执行状态有两个取值状态（“变量”） `V1` 和 `V2` ，分别可以取值 `v` 和 `e` ，也可能是 `?` 。

`E1` 为 `P` 的初始状态，满足 `E1 = {{V1, V2} | V1 = v}` 。以 `→` 表示映射，那么有：

复制操作 `C: E1 → E2` 满足 `E2 = {{V1, V2} | V1 = v ∧ V2 = v}` ；

转移操作 `M: E1 → E2` 满足 `E2 = {{V1, V2} | V2 = v}` 。

直观理解，一个变量的值在复制后出现在原来的变量和另一个其它变量中；一个变量的值在转移后出现在另一个其它变量中。

2.结构化复制和转移

为了在语言中表达复制和转移，需要借助类型系统及约定的原始操作。

对于复制来说，除了少数“纯函数式”的语言，基本的操作是赋值（本身就是一种复制）。相反，可能由于复制可以用转移和赋“空值”复合，一般语言不直接提供转移操作。

实际上，赋值作为简单的复制一般是不够用的。一般来说，语言提供的抽象手段允许复用基本操作，对于构建复制和转移操作也不例外。和经典的函数调用类似，这里的有效的复用可以是结构化的：递归的且有层次性的。

C++ 提供了复制构造表达一种符合结构化程序思路来构建复制（初始化）语义：自定义类型的复制使用成员复制（以及构造函数体的其它副作用），直至基本类型的复制（同赋值）。自定义类型的复制构造反过来也成为了复制赋值的基础。

C++11 的转移构造也使用类似的机制。而对于基本类型来说，复制和转移在这里是一致的。

从上面的形式可以看出，复制实质蕴含转移：任何复制都是转移，复制是附加了更强约束的转移。

逻辑上转移的实现似乎可以用复制和删除取代，或者扩展转移机制以实现复制。看起来似乎提供复制和转移两套近似的机制有点浪费。那么：

a. 复制能代替转移吗？

答案是可行的——事实上 C++11 前就是这样做的。但是 C++ 在这里有一个特性——复制构造允许复制成员以外的副作用——导致了致命的弱点：复制开销无法被轻易优化掉。为了克服这点， ISO C++ 甚至有专门的语义规则允许忽略复制的副作用。区分复制和转移允许更好地减少不必要的开销。在逻辑上也有更重要和普遍的漏洞。原理见下文。

b. 转移能代替复制吗？

让转移成为更“基本”的复制，扩展转移以实现复制至少对于 C++ 来说也不符合实际。复制无法被转移直接取代，除非允许重载基本类型的操作——如果是这样，基本类型的操作又用什么表示呢？在 C++ 的框架内部行不通，而暴露底层实现（如汇编）给用户看来也不是干净的做法。

3.控制流中断：异常安全

比较 `{{V1, V2} | V1 = v}` 、 `{{V1, V2} | V1 = v ∧ V2 = v}` 和 `{{V1, V2} | V2 = v}` ，可见复制操作和转移操作在这里有一个根本的不同：复制操作需要多维持一个状态（“变量”）。

从体系结构上（注意，并不限于具体语言的实现）来说，若不限定状态在语言实现上的某一个 `phase of execution` （普遍地，如编译时），复制操作必然需要申请获取资源（典型地，如内存）；除非预留（本质上来说和限定 `phase` 类似，所谓“静态”的优化），这种资源的分配总是可能失败。

一旦失败，依赖于被复制的状态的操作就不能进行，即错误必须被处理；否则逻辑上就不符合期望了——也就是错的。由于基本操作的普遍性，让控制流中断而不是让用户在调用端手动检查是比较简便的做法：在 C++ 中即抛出异常。（因为允许失败以及析构函数保证的决定性生存期终止语义，“变量”—— C++ 的对象可以很直接地作为资源的抽象，即 RAII 惯用法。）

上面提到过的复制和转移是结构化的，这意味着除了基本类型（复制和转移一样，没有异常）外在任何层次都得考虑这个问题。只要有一层复制初始化操作（复制或转移）可能出现异常，那么依赖于这个不保证无异常的操作的复制初始化都不能幸免。也就是说复制可能抛出异常这点在一般意义上是普遍的（注意C++03的实践已经证明了限制异常规范的做法的失败）。

而转移操作没有这个限制。理论上来说，任何转移操作总是可以被设计为保证成功的。所以转移操作往往用 `noexcept` 限定没有异常；标准库的泛型组件一般也要求被操作的对象类型可无异常抛出转移。

考虑到异常安全和无异常抛出保证实现的困难性，这个差异是如此重要和普遍，以至于即便无视复制初始化的性能问题（毕竟还可能让 RVO 等挽救一下），单独区分出转移也是相当有意义的。

4.转移和交换

注意 `M: E1 → E2` 中， `E1` 和 `E2` 是对称的。这意味着两点：

a. 转移本质上是一种（对称的）状态迁移；

b. E1和E2作为执行状态可以置换。

对于 C++ 来说， a. 的意义在于如果转移可以用简单的操作实现。注意对象转移后仍然会被销毁。对于基本类型对象，因为没有有副作用的析构(non-trvial destructor)，所以可以直接复制/赋值（这再次说明了基本类型复制和转移的一致性）。

不过对于整体状态来说还有个陷阱，导致程序不一定总是能得到期望行为。 `{{V1, V2} | V2 = v}` 隐式地蕴含 “ `V1 = ?` ”，即被转移的“变量”的状态是无法保证可预期的。如果需要可预期（仍然能足够安全地使用被转移了的资源），那么简单复制就不一定行得通了。比如说一个预期保持所有权、通过 `delete` 释放的内建指针，如果复制代替转移而不清空，最后会导致多次 `delete` 。　　很遗憾这里C++无法有效地提供安全检查。一种安全的惯用法是使用交换例程（如 `std::swap` ）来实现，不过需要指定的类型可交换。这早就是一种异常安全代码的常见技巧——C++11也约定 swappable 要求无异常抛出。

b. 是 C++ 语言层次上难以利用的特性（和复制构造类似，转移操作也允许存在用户自定义的副作用，省了转移的优化也是受限的），在此不讨论（嘛， Prolog 什么的就算了……）。

5.转移语义：显式实现 vs 语言特性

一个语言可以不直接支持转移需要的特性，而只是支持间接操作的“引用”类型，这样仍然可以实现转移。这时候典型的转移实现被称为“浅复制”(shallow copy)，区分于完全值语义的（递归）“深复制”(deep copy)。

对于语言设计来说，这可以使设计简化，而不必像 C++11 那样引入右值引用类型来折腾类型系统。不过，这样往往意味着用户需要完成更多的重复代码。至少对于 C++ 这样对象模型中显式声称“对象即存储”精神（继承于 C ）来说的语言来说是这样。

从实际需求出发，理想情况下，语言应该能够分辨出转移和复制这两种不同的需求。像 C++11 在类型系统上做的手脚已经被证明可行的，那么除此之外，有没有比让用户显式实现更简单的方法呢？上面说过，至少在 C++ 中复制和转移无法互相取代，不过如果不限 C++ 嘛——

a. 转移取代复制。要是提供允许用户指定行为的基本类型操作的公开接口，那么转移就可能直接作为复制的基础了。不过，因为C++仅有的静态分派和多态（重载）依赖于类型系统，像C++要这么搞恐怕类型上的折腾还是省不掉……而实际上，分派也好模式匹配也好需要的签名要是类型以外的其它新特性，本质上来说也属于传统类型理论的范畴内。

b. 复制取代转移。不使用显式类型，取消复制初始化的副作用，让语言实现预测复制到底需要怎么实现。这是常见函数式语言的做法。这种复制相对很容易优化掉，代价是实体（“变量”）无法隐式地作为具有状态的资源抽象。

至于 C 、 Java 什么的……算了，老实人肉实现吧。

所以 C++ 其实也算个典型，虽然看上去糟烂混乱不堪大用，不过小心点用也还算差强人意了。

6.结论

坑掉算了。

正确地黑C

Created @ 2014-07-29 09:01, rev v5 2015-04-12, markdown @ 2015-09-14.

本版暂时当作提纲，不做详细展开讨论，以后可能更新。

注意：本文主旨不是政治正确。

1.设计

C 的设计相对于同期来说是局促的。 C 语言具有明显的历史局限性是不争的事实。

类型系统是一个显著的例子。理论上， typed lambda calculus 在当时（ 70 年代）即便没有流行也已经有了数十年的发展，但是 C 的设计并没有有效利用当时的理论成果，还在前人的经验上开洞（比如奇葩的函数指针转换）。

类似地，数组类型也不是第一类实体，也会有类型上的修正。

这些无聊的设计除了让抽象变复杂，限制用户的自由外，唯一的好处也许就是顺应之前的具有局限性的语言实现的习惯了。不过，今非昔比，这样的设计并没有简化现在的语言实现——不管是C还是其它语言。

这种仓促的设计影响深远。 C++ 至今仍然把一些肇始于 C 的奇葩转换保留在“标准转换”中，以至于重载的规则加倍复杂——考虑到兼容问题以及维护规则本身的困难，这些无谓的复杂性今后可能永远也无法移除。

左值(lvalue) 是一个不幸的设计。一开始作为文法性质，并没有考虑到潜在的复杂，以至于后来引入 `const` 后，功能和区分左值在相当大范围内重复了（若完全一致反倒还好点）。这点另外有说过，按下不表。而这里重要的是，也无法指望丢弃冗余性（ C++ 甚至变本加厉）。

当然，考虑历史的主干， C 来自于 B， 来自于 BCPL， 来自于 CPL， 来自于 ALGOL 60， 来自于 ALGOL 58， 来自于 FORTRAN 。这些语言都没有这方面的合理经验积累。所以C的设计的局限，并不难想象。

也有一些其它的不足之处——丢弃有用的特性导致的倒退。不过，有些被纠正过来了，例如 BCPL 后扔掉的单行注释（//），被某些 C 方言、 C++ 和C99以及其它一些 C-like 派生重新吸收。

还有一些奇葩的莫名其妙的地方。B里面叫vector的东西说的好好的，怎么到C里面就变成array了呢？只是因为加了个残疾的“数组类型”？（B是没好意思说成有类型的……）为什么array而不是vector就非得能对元素“O(1)访问”——这种偏见是谁发明的？（另一个不良后果是

A.Stepanov搞STL的时候没词了就顺手捡了个……）

2.标准化的有效性

不能否认，在被规范化的语言（而不是实现）针对平台的可移植性来讲，C差不多应该是现在的语言中最强的了。ISO C++ 在某些少数极端情况下的可移植性的确不如 ISO C（见 WG21/N4049 ）。而其它语言，假定 1 字节不等于 8 比特的大小就足够打退堂鼓，若不够可以再加上允许原码/反码作为有符号数表示——这两者是 ISO C 和 ISO C++ 都明确支持而其它大部分语言规范都与之矛盾的东西。

标准化的一个重要作用是取得共识，避免一些重复工作，提升可移植性，减轻用户（包括实现者）的负担。若标准被架空而没有被实际使用，标准本身就失去了绝大部分意义。

这里的一个反面素材是 C#， .NET的实现都出到 5.0 了， ECMA-334 到现在还是当年 2 的版本……（还有 C# 里把 finalizer 叫成 destructor 的奇葩导致混乱的说法在 ECMA 里被纠正了， MSDN 上仍然将错就错 。）

啥， ECMA 是区域性组织，不够权威？——人家现在叫 ECMA International 好不好。不过不够权威好像是能坐实点， C++/CLI 在 ISO 标准化被英国的意见驳回 ， ECMA 就通过成 ECMA-372 了……

那么这里就拿 C++ 比较好了。同样是 ISO/IEC JTC1/SC22 下的工作组，两者的产出看似类似，效果大相径庭。

而敢无视 WG21 的实现——据我所知，一个都没有。即便 GNU 早年隐晦地表达过和标准划清界限，现在来看在 C++ 前端是口嫌体正直了。反观 GNU C， 这个效应就弱得多。

简而言之，ISO C虽然整体上是有效的，但是对于语言实现者来说，效力略为不足。

概括起来，这有两方面原因。

其一是 WG14 本身的活动没有WG21强调“open”。 WG21 的文档 早就被公开多年，而 WG14的 没记错的话到2012年才被公开。

可能是由于 C++ 本身的复杂性以及历史教训（轻率引入 `export` 和dynamic exception specification）需要避免消极的 design by committee 的影响，参与 C++ 工作讨论的非委员会用户活动非常显著， Google Groups 里讨论标准提议的论坛 已经开设了好几年。 isocpp.org 也是一例。 WG21 甚至在 github 上拥有公开仓库允许用户 pull request 修改标准草案的内容。

反观C方面呢？一个对应的东西都没有。

作为工作产出来看， WG14 公开的 paper 也要比 WG21 少得多（得多……）

C++ 比起C真有复杂到这种程度么？还是说 C 的“社区”（暂且这么说）在传统上就“不够进取”？或者根本不愿意达成共识呢？

第二点恐怕未必。 POSIX 就比较活跃了。

可笑的是，维护 POSIX 的 Austin Group 和 WG14 之间也能出现琐碎的意见分歧。参见 WG14/N1174 。

原因是其二——还是 C 的设计。

C 欠缺了太多东西。

上面的隐晦分歧就是指，是不是在标准化的接口中，允许函数返回动态分配的对象然后让用户自行释放？ ISO C 的意见是“不”——于是 `asprintf` 乃至 `strdup` 都不可能出现在 ISO C 标准库，但这个原则之前看来从来没有被正式提起过。而 POSIX 方面以及其它传统 C 用户大概不会这么认为（特别是 GNU ）。

其它主流语言里还有这种奇葩问题么？

而 ISO C 新引入的东西，很多也不是自身的设计。

ISO C11 引入的 sequenced before 的 wording ，是 WG21/N2239 提出来的。注意，是 C++ 的 paper ， C 后来照搬过去了（与之相关的还有并发模型 ）。

（嘛， C++ 比较激进删除了之前ISO C引进的sequence point，不过这里有个bug，漏了 sequenced after 这个定义，我邮件过去了，处理情况见这里 。）

C11 同时照搬过去的还有多线程和 atomic 的基本概念。由于语言特性上的先天不足， C 没法做到 C++ 的优雅（虽然这词普遍恶心，但用在这里的确不错）。（看看那个奇葩的 `_Atomic` ……）

C11 还不得不引入了某种意义上的“临时对象”， 这种手段又是埋坑 。

C11 绝无仅有的东西呢？哦，比如 `generic-selection` ？我问一下，这里谁对 `_Generic` 有印象的？知道它是怎么回事的？能说清解决了什么问题，并且是怎么起作用的？有多少人在实际代码中用过？在其它语言中类似的东西是什么？

ISO C 比 ISO C++ 多出来的，特别是近来新添加的，差不多尽是广大C用户都难以认同的琐碎玩意儿……

WG14， 请不要玩脱。

3.用户素质

本来我在这里不想攻击任何人。但是C的用户在辩护语言和实现表达自己的观点时，表现出来的槽点明显比其它语言的用户多得多，并且不少自以为得计，优越感爆棚。忍无可忍。

在这里起到负面影响的用户都可以概括成不同程度的“不懂装懂”“在不熟悉的领域里瞎 BB ”“误导”，不过可以分成那么几类（也有不少复合的）：

a)原教旨主义

认为C是程序设计语言发展历史上的“正统”——却连老祖宗ALGOL的地位和影响都不知道，甚至和亲爹B语言之间的差别都一问三不知。

b)盲信

不管可以引证的依据和理性思辨而盲目认同一些经不起推敲的观点（比如C比起其它高级语言总是“性能高”“开发效率低”）。

c)无知

缺少其它语言的使用经验却臆测行为和实现。甚至对C自身的基本概念（比如“对象”“左值”“未定义行为”）都说不清楚。

d)不独立思考

缺乏怀疑精神，对符合固有印象的说法来者不拒，却不考虑理由。“我听说”……“人家××× 就是用 C 写的！”

e)缺少专业基础

没有 PLT 常识，对其它语言品头论足，对其中和 C 设计不同之处——而不是不足之处做出非理性的批评。

从经验上来看，这些用户中最核心的部分同时是 UNIX 的脑残粉——包括一些只用过 Linux 然后把自己包装成 UNIX 粉的。

这些用户，绝大多数都说不清楚 C （也许还有 UNIX ）历史和发展方向之类的详细问题，要提一些现有实现的缺陷和可改进之处都支支吾吾，甚至说不清楚为什么好用，满足了什么需求（某些果粉都比他们更强一点）。

脑残粉本身不足一提，不过当一些“知名人物”也具有这些特质之后，他们就好像找到了什么被撑腰的了——却不知道很多“大师”在评论这里的问题上很多也是半外行，跟普通用户无异。

4.专治各种不服：

有谁自认为对 C 本身了解更深刻而全面反对以上观点的（特别是挂名在 WG14 内的——有么？），欢迎对号入座战个痛。

增补 1 ：

Appended @ 2014-07-29 15:58.

既然提到了有符号数问题，这里加列个草稿。虽然算不上 C 的问题，但不少 C 和 C++ 用户都仍然对这个问题稀里糊涂以至于设计出渣接口。

对于 C/C++ 以及其它大部分语言来说，不管是有符号数还是无符号数，首先指的都是刻意设计用硬件上具有特定表示的整数数据类型，即有存储空间大小和范围限制的、值的表示连续的定长整数。 从类型系统的角度上来说，对于表示算术操作的类型而言，类型限定它有意义的取值范围，同时可用于决定它具有的操作。

这里的整数类型和数学中的整数显然不同——最显然地，它是有限集。但关于操作上的不同就经常被有意无意地忽略。大多数情况下这不是被期望的。

实用中，有符号数和无符号数的不同在于，无符号数具有确定的表示方式，且尽管在不同的体系结构上大小不一致，但都遵循2^n的模算术。注意和数学中的整数的一般算术操作的差异，例如模算术减法在不够减时结果会回绕(wrapped) ——比被减数更大。

而有符号数则没有这样普适的一致性：至少可以有原码、补码和反码表示。 ISO C 和 ISO C++ 都规定：允许有符号数使用三种表示方式（之一）。占用空间相同的有符号数的表示不保证可移植性。同时，有符号数的操作导致超过表示范围（溢出）则行为未定义。而对于有符号数的一些转换和位操作是实现定义行为。 而与此相反，无符号数的性质要确定得多。在确定模算术的意义上可以推导出一些方便的性质，例如无符号数一定不会溢出。无符号数也不会有上述有符号数操作的实现定义行为导致的实现差异。 从实现角度上看，通常无符号数更容易被硬件高效地实现。只支持有符号数而不支持无符号数操作的实用体系结构似乎从来就不存在。

从抽象角度上看，无符号数的上限比同样大小的有符号数更大，所以在确定不需要符号时，更有利于正确的数值表示。

因为上述原因， `sizeof` 的返回值 `size_t` 类型确定是无符号数类型。类似地，在 C 和 C++ 中，除非必要，应当避免使用有符号数。

相对地， Java 使用（范围被硬编码的）有符号数而放弃使用无符号数。这在一般意义上是一种错误的（类型系统）设计：损失功能且带来了更多的麻烦。

除了失去上述无符号数确定的性质（ Java 倒是确定了有符号数位操作的语义），这导致一些只需要无符号数的场合，同样的大小范围损失了近一半，并且导致程序更加不清晰：用户没有办法表示“我这里就只需要一个非负数”“你应该能预期结果是个非负数”（天杀的 Java 还没 `typedef` 和宏……）。

只有有符号数导致 Java 需要引入 `>>>` 操作符以便区别对待符号位，在另一方面导致了语言的复杂。缺少无符号数还导致一些应用上的麻烦——例如实现随机数生成器、通信协议或者图像处理算法之类。

Java 抛弃有符号数的理由据信是“无符号数更复杂，因此不需要”——根据上面的分析，这是典型的扯蛋。一个比较可能的实际理由是对无符号数“莫名其妙”地回绕的无谓恐惧，特别是对结果比较操作上。但事实上，任何一个对这些语言中算术操作有清楚了解的用户，都不应该陷入这种陷阱，特别是编译器能轻易检查出对有符号数的不保证兼容性使用的情况下。

大概也正是因为如此， C# 没有在这里“借鉴” Java ，还是补充了 `uint` 。不过， C# 的语言规则导致 `uint` 和 `int` 诊断消息设置有些神经质，大量用户代码中还是充斥着本来不应该出现的 `int` 。设计者应该只是确信 `uint` 不可少，但只是有符号数的补充，而非尽量鼓励使用的对象。这和 C/C++ 的设计非常不同，也并没有完全发挥无符号数的好处。

于是话说回来，为什么在 C/C++ 这样支持并且某种意义上更鼓励尽量使用有符号数的语言中，为什么还是有人非得喜欢 `int` 到底呢？

恐怕原因和上面的一致。简而言之：这些人（包括某些抽风的语言设计者）自己就没把这种受限的定长整数算术操作搞明白，又欠缺谨慎、耐心和全盘考虑，可能还有些偏执。

增补 2 ：

Appended @ 2014-08-11 16:53.

关于C/ C++ 和汇编语言的关系。

1. C 是 C， C++ 是 C++， 汇编是汇编。

ISO C规定了允许 `asm` 关键字嵌入汇编代码作为扩展，但 C 并不依赖这项特性。

可能由于 Bjarne Stroustrup 等对阻止语言分裂的立场（如果需要另一个例子，可以参照对 Embedded C++ 的观点），ISO C++ 没有 ISO C那种“扩展”的概念， `asm` 关键字直接是正式的语言组成部分。

C++ 的 `asm-definition` 的含义是实现定义的，但同样也并不见得里面就是汇编语言——例如， Cheerp 使用 `asm` 内联 JavaScript （虽然后来改为了 `__asm__` ）……

不管是 C还是 C++， 作为实例的不支持内联汇编的实现也不难找，例如用于 ARM 和 x64 的 VC++ 。

所以即便是 C， 指望和汇编互操作在语言层面上缺乏一般的可移植性。

2.作为实现的汇编语言和对象语言（例如C或 C++ ）的关系。

一个常见的误解是“ C 和汇编对应”。事实上这种保证根本不存在。对于其它高级语言也大抵如此。 尽管现实大多数 C 或 C++ 的实现使用汇编作为中间表示，没有谁强制所有实现都遵循这一点。于是这里需要排除不使用汇编作为中间表示的语言。也就是说，要考虑这点，就不是讨论 C 或 C++ 语言本身而是具体实现了。

需要肯定的是，对于某个实现来说，作为中间表示的汇编和 C 和 C++ 的某些操作的语义接近，有一些现实意义，主要在于互操作性的清晰简便。

3. C 和 C++ 在这里的差异。

注意，对于使用 C 和 C++ 的用户，上述需求都存在并可以实现。 C和 C++ 实现在此的主要差异在于， C++ 提供更高级的抽象，这些操作在 C 中往往没有对应，所以这部分在这里没有比较的意义。 从实现的复杂性来看， C 和 C++ 公共具有的一些操作往往比较能直观地被理解。 C++ 的其余部分涉及到更多实现细节处理起来较为困难。如果使用这些操作，的确会导致一些现实问题，特别是 C++ 实现的 ABI 往往比 C 实现的更混乱。

但是，在需要考虑这类需求的场合，使用什么操作是可控的。如果只使用 C++ 和C中类似的特性，那么问题实际上差不多复杂。

差异只是用户对 C 和 C++ 的实现（即便排除 C++ 中实现起来看似不够直观的部分）理解有所不同而已。

也就是说，到底还是用户自己的原因。（不作死就不会死……）

至于运行时依赖问题导致 C++ 比 C 在这里欠缺可用性，这是另外的话题。

4.还要注意的是，上面这些到底只是“接近”，并不是所谓的普遍存在的“对应”。

这个问题细说起来可以展开很多……

有一个目的上出发的根本观点：设计可维护的程序，依赖的应该是接口而不是接口的实现。依赖实现的 hack 只是妥协而不应作为可靠的常规手段。

对于能够使用这些接口解决的问题，不需要也不应当依赖它们的实现。

一般地，高级语言的规范自身通过描述程序的语义和/或行为，事实上给出了一套更高级普遍的接口。 ABI 规范作为适用于二进制互操作的低层次的接口，相对来说提供的是语言规范的实现，以缩小适用范围为代价，补充语言规范中没有限定的一些细节。

C / C++ 这类本机实现的 ABI 一般可以分为体系结构和编译器相关的两部分。可惜某些实现不给出清晰的、友好的、公开的规范，拿黑箱让用户自己猜……活该谁倒霉呢。

没有足够能参照的 ABI，所以需要互操作时，不得不通过观察生成的中间代码来预测实际可能的程序行为，这也不是不能理解。这种不保证可移植性的程序行为，语言规范更加管不着。

（这里中间代码说的就是汇编，其它类似的还有 JVM bytecode 、 CLR IL 之类的中间表示，不过传统上汇编占这种情况的绝大多数。）

但在二进制互操作以上的层次，仍然依赖上面的手段——无理由、无益地依赖实现——就有些匪夷所思了。 或许实际情况是，这些开发者根本就对这些原则性策略缺乏清醒的认识。

造成这种情况的理由可能就是在于方便依赖“和汇编对应”之类实现细节在作怪——至少表面上来看，只要“会”汇编，对生成代码有个大体的经验，很多时候就能弥补对不清楚接口导致无法预料程序行为的不足了。 这显然是类似“撞大运编程”的错误姿势。只是大部分这类用户甚至不会考虑体系结构之间的可移植性，所以这种错误认知也不容易起到负面影响。一旦有机会，坑的会是谁呢。

由于种种历史原因（比如说早期编译技术比较初级导致编译器一般不会进行复杂的变换优化）， C 的实现往往给人一种在接口规范之外也容易找到“对应”的印象。不过，现代的实现对此已经渐行渐远了。 就这点这当然不能怪 C ，那么。这种破事说到底也只能当作某些误导作祟导致某些用户技能树点错学岔了产生误会了。

此外，对于学习语言来说，这种错觉的危害一般更大。考虑到这里的问题，先学汇编再学C这种路线应当被慎重考虑，不推荐一般用户以免形成先入为主的观点。当然，这是另外一个问题了。

5.附带的其它结论。

除了上面的需求导致的现象外，一个副产物是， C++ 的抽象让不善于使用正确姿势思考问题的新手望而却步了，其中不少可能就学看起来“底层”的 C 去了……

对于 C++ 来说，的确可以算 teachability 的耳光响亮。不过，在“底层”方面的逗比用户也因此更少了，也许也是好事。

（另一方面，“纯 OOP ”逗比用户被 Java 之流给架走了不少……不过剩下来也许还有两方面都特别逗的用户，那就不怎么样了 wwwwwwwww ……）

增补3：

Appended @ 2015-08-09 10:00.

突然一时不爽，就拿 C 和 C++一起黑。当然，还是 C 的锅。

总体属于设计问题：没理由的不一致。

具体例子有很多，不过 C 就有并且 C++照搬的垃圾设计就稍微少一点了。这里值得一提的是逗号表达式——体现无能然而搞得莫名其妙。

逗号表达式有鸟用？其实说白了就是在本来该写表达式的地方想硬塞下语句而已。严格地说，意味不明地区分表达式和语句这种烂设计并不限于C而是某些Fortran/ALGOL系的硬伤，但是在C里还有特别恶劣的其它影响：搞得规则不伦不类。逗号前后的表达式是不是能确定求值顺序？因为存在逗号表达式，答案是不一定。可能有人说把其它用逗号的地方（像是函数参数列表）都改成从左到右就一致了，然而这样就损失了“放弃明确顺序而让实现自由优化顺序”的表达能力，和Java之流一样蠢了。所以锅在逗号表达式顺序自己。一旦允许表达式套语句，逗号表达式就是鸡肋。事实上也有这样的扩展——GCC的语句表达式。

在 C++ 中，由于逗号可以重载，却不提供操作数求值顺序保证，这种混乱更加明显。所以考虑到误会的风险以及简化问题，一般回避重载逗号，甚至于泛型算法实现中假定不存在逗号重载（然而大多数用户是直接无视了）。但这也就是无法直面从C继承的锅的鸵鸟政策。另外，有 lambda 表达式后，非重载的逗号就更加鸡肋了——即便是不重载的 C++ 逗号表达式，因为更加不像C那么“有用”，所以更罕用，在更复杂的语言规则中导致误会的风险也更大。

上面说了C（ C++ 也一样）允许刻意不限定求值顺序。然而这并不是直接好用的语言特性。 YBase中 的 ydef.h 提供了宏 `yunseq` 用函数模板包了一层才稍微清楚，然而对于 `void` 表达式无能为力。而如果逗号表达式一开始就一如函数参数列表一致地提供非限定顺序求值的语义（至于需要限定字面顺序的地方，就用“语句”——比如说，使用分号替代逗号），那么本来就不需要这样费事了。可惜事实不是这样。

3.1

Appended @ 2015-08-09 10:04.

看起来逗号表达式如何“有用”，也算一般理解了。

3.2

Appended @ 2015-08-09 10:07.

`yunseq` 的实现（ C 就别想了）。

增补4：

Appended @ 2015-08-09 10:54.

还是关于烂用户的事——但也不全是用户的问题。

用户理解得是否正确清晰而不会造成破事，除了语言设计，语言规范的质量自身也是重要参考因素。 首先需要澄清，我并非觉得 ISO/IEC 14882 作为技术文档的质量很好——相反，它相当罗嗦，有不少 bug （虽然被我和其他少数人不断修正但也不断会窜出新的），并且总体来说并不怎么容易看。

而这些不足也同样适用于 ISO/IEC 9899 ，除了不断窜出新 bug 这点（嘛……理由是 ISO C 更新实在太少）。

当然，这两个都够吊打体例不够清楚到技术标准程度的 spec 了，比如 JLS （现在的版本 甚至比ISO C++ Clause 17 之前的部分还长，以后我倒是看看谁说 Java 语言规则简单然后就能啪啪打脸了）和 M$ 的 C# spec （ ECMA 那个还好不过显然过时放置 play 中）。

但是，虽说这两个很多地方都不怎么样，一些关键的地方还是有高下的。

比如说，关于重要的程序执行模型，也就是 ISO C 放在了 Clause 5 ， 而ISO C++ 放在了 Clause 1.9 的内容。（且不说C11在此照搬了大段 C++ 11的基本概念）尽管前者定义清楚了一个实用的 observable behavior 的概念，后者的内容比前者还详尽得多。这导致了关于 conforming 问题上后者更清晰。

一个比较直接的实例是，关于未定义行为“在什么时候发生”的破事。

实际上这里即便是 ISO C ，本义也是清楚的。但是就会出来这种逗比问题 ：

> 这是一道精妙的题目还是一道傻逼题目？
>
> 问题出自：大型互联网公司为什么会出一些质量低下的校招题目？
>
> http://pic4.zhimg.com/9dc0fadbb39515914415411f958a8cb7_b.jpg
>
> 有“我们微软”的RSDE vczh 称：
>
> “但是就截图第七题来讲，我发现只有A选项有可能得到4，其余三个选项无论选取哪种顺序都得不到4，很屌。这是一道精妙的题目，不叫傻逼题目。你们不要看见
>
> 考++就说这道题不行，++虽然有undefined behavior，但是这道题目的考点是你知不知道undefined
>
> behavior下面能允许的东西其实也是有限的，懂不懂得穷举他们。”

这几乎是常识性错误。vczh的脸自然（？）已经被回复打肿了（只是看了zhihu的尿性所以全文按合理使用的形式引用，粗体略）：

> 张景旺，图灵的机器上，能否承载邱奇的梦想
>
> 呵呵一笑百媚生、舒夜、知乎用户 等人赞同
>
> 不同意你们的说法，题目改成“可能为４”的话，那么４个答案都是正确答案。undefined behavior出现什么结果都是可能的。
>
> 题目改成：在我们微软的编译器编译之下，下列哪个表达式的结果为４？就是精妙的题目了……
>
> 或者更直接一点：
>
> 你是信Ｃ＋＋标准？还是信ＶＣ＋＋？
>
> Ａ：信ＶＣ＋＋
>
> Ｂ：信标准
>
> Ｃ：我要进Ｃ＋＋委员会，将来我说了算
>
> Ｄ：老子编程从不想标准还是编译器啥的，生成可执行文件就是成功……
>
> 这样就是一个绝对精妙的问题了！这样正确答案就很明显了，而且还能筛选出“我们微软”想要的人才，毕竟这三十年来我们微软也以无视各种标准而出名……好传统，要保持！

引用这里的一些评论。首先是垂死挣扎：

> vczh
>
> 显然undefined behavior是有范围的。多个++出现的时候，标准说得很清楚，尽管++发生在什么时候不知道，但是有多少次++就加多少次1是肯定的。因此就算你写a+++a++，a也不可能从0变成3。
>
> 2014-10-13

嗯，我就先不管“标准说得很清楚”在哪了——虽然其实这就是就这个例子我之后要讨论的内容。

紧接着被直接打脸：

> 薛非 回复 vczh
>
> 显然undefined behavior是有范围的。//无知
>
> 2014-10-13
>
> 薛非 回复 vczh
>
> 多个++出现的时候，标准说得很清楚，尽管++发生在什么时候不知道，但是有多少次++就加多少次1是肯定的。//臆测
>
> 2014-10-13
>
> 薛非
>
> 因此就算你写a+++a++，a也不可能从0变成3。//妄断

下面没存在感的附议略。

看客也有明白人，虽然引用的论据强度不够：

> 2014-10-13
>
> Jimmy shen
>
> In computer programming, undefined behavior refers to computer code whose behavior is specified to be arbitrary. It is a feature of some programming languages—most famously C and C++ .
>
> 这次轮子哥估计要跪了。坐等二位大战。
>
> 2014-10-13

不过稍微有一个需要注意一点的（连续三个评论，“知乎用户”应该是两个人，下面标记为A和B）：

> 知乎用户 回复 薛非
>
> 我觉得您的IQ可能需要充值了，我特意去翻了下手头的2003版ISO，如果是C或 C++ 的话，这个所谓的undefined behavior要保证不与前面的运算符章节冲突，只存在两种可能……
>
> 2014-10-22
>
> 知乎用户 回复 知乎用户
>
> 我觉得您的IQ可能需要充值了，你根本还不知道什么是undefined behavior
>
> 2014-10-25
>
> 知乎用户 回复 知乎用户
>
> undefined behavior再undefined也要符合标准在其他地方对这个运算的限制。只强调一个undefined却不看其他限制条件，这叫玩儿文字游戏。
>
> 2014-10-26

vczh的脸我不感兴趣（看了这个问题我正再次考虑他是不是有资格和方是民和王垠等人相提并论），然而最后一点需要澄清。

ISO/IEC 14882:2003

> **1.9**/5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible execution sequences of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution sequence contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

看括号里的。好了，“特意去翻了下手头的2003版ISO”那位的脸已经肿了。

虽然上面没有先吐槽翻了一下2003版的ISO如何得出C的结论，不过翻了下ISO C的确是没有那么方便打脸的条款的（当然，这个问题的结论不变）。

不管怎么说，翻差不多同样晦涩的ISO C打脸却不方便，收益不够，相对于ISO C++ 来说也算是个黑点了。

C++ 设计缺陷

Created @ 2014-08-01 21:46 rev2 2014-08-01 22:14, rev3 2015-09-15, markdown @ 2015-09-15.

0 不识时务的妥协

特别地，兼容（而不是取代） C 。

1 类型系统

设计上烂得最广泛的毫无疑问是类型系统——特别地，缺少很多本来该在类型系统内的东西。

1.1 基本类型

基本类型在（所谓的 *fundamental types* ） C 的基础（所谓的 *basic types* ）上重写了一遍，有微妙的不同（如宽字符类型），但换汤不换药，缺点差不多一个没少。

1.2 标准转换

继承了 C 的低质量的转换：

- integer promotion
- array-to-pointer conversion
- function-to-pointer conversion

1.3 非一等类型(first class types)

继承了 C 对数组和函数类型的差别待遇。除了上面的转换，作为在函数参数和返回值也有反直觉的限制。

1.4 动态类型(dynamic types)

添加了动态类型却支持有限，实际上搞出了 ABI 的坑——缺乏公开规范，乃至同一个体系结构和操作系统的二进制代码都没法兼容。

题外话，为了描述规则的清晰性 ISO C99 添加了有效类型*(effective type)*，实际上相当于动态类型，不过没有明确的运行时支持。这反倒不影响实现。

1.5 对象类型

1.5.1 语义

和 C 一样， C++ 的对象*(object)* 指存储。和 C 不同， C++ 对象的销毁同时可能通过非平凡析构函数*(non-trivial destructor)* 的调用而具有副作用(side effect) 。

但事实上，要求存储和要求生存期边界引发副作用是正交的特性。 C++ 显然没能处理好这一点：为什么一个只关心析构副作用的对象仍然必须占据存储？

此外，类类型是对象类型。不使用特殊规则，基类或数据成员子对象将会占据存储——即便实际上没用到。因此语言不得不使用特别规则进行空基类优化*(empty base optimization)* ——而且对空数据成员的位置还有限制。考虑到许多时候这里的类只是和静态类型系统交互，这种冗余的语义本应是毫无意义的。

1.5.2 分类学

C 的对象类型包括 `void` ，而 C++ 把 `void` 独立出来。但是，仍然有一些容易混淆之处：`void*` 是 *object pointer type* ， 但不是 pointer-to-object type 。

1.5.3 布局

添加了和 C 不一样的布局，却不提供足够的、明确的、可移植的支持，导致基础设施的失效：

- 如 `offsetof`

ISO C++11 提供了标准布局类*(standard-layout class)* 的概念，基本继承于之前的 *POD(plain-old-data) class* 。然而这个概念的定义在之后版本的标准中仍有微妙的变化，并不容易被理解。

标准布局类型*(standard-layout type)* 的概念构筑在标准布局类之上。其它对象布局就没有附加约束了，如果需要可移植的 ABI ，几乎只能使用标准布局类型。这排除了许多特性，例如具有虚函数而不满足标准布局要求的多态类*(polymorphic class)* 。

1.6 值类别(value category) 和引用

添加了引用类型，想取代 lvalueness （参见 WG21 最早公开的 paper ），后来却坑了。

在表达式求值的特殊规则下，引用类型的表达式被视为被引用的实体。这种特殊规则减少了那么一点其它一些语言规则（如 ADL ）的描述，却实际上使引用类型也不完全是一等实体了。

- 现在还越搞越复杂，搞出了多种引用类型和 value category ，依然不视为类型
- 并且语义和用例容易被误解，且很多时候难以避免冗余编码

1.7 添加了参数化多态

却对高阶参数化类型（模板模板参数）有莫名其妙的限制导致不可用。

1.8 添加了实质上的参数化类型

却独立于名义类型系统(nominal type system) 之外。

概念(concept) 有望部分地改善这点，但无法改变基础设计分裂的现状。

1.9 添加了类型推导(type deduction)

却不足够支持一般意义的类型推断(type inference) ——例如在构造函数上不起作用，需要额外的工厂方法(factory method) 作为惯用法。

1.10 添加了实质上的重写系统

却没有完善的类型系统支持，如：

- 基于类型的模式匹配
- 自定义重载规则

2 一些深刻的设计失误

是差不多所有Algol60直系后裔都存在的功能缺失。

2.1 同像性(isomorphism)

这导致语言中不得不对反射做出特别对待。偏偏到现在还没有。

2.2 底层高级流程抽象

具体点说，能代替 J operator 或者 `call/cc` 之类的东西。

这同时导致诸如 coroutine 等依赖基本控制流的抽象不能在不添加语言特性的前提下被可移植地高效实现；同时也有下面的异常问题。

2.3 活动记录抽象

这导致典型实现及其用户过于依赖特定于体系结构的运行时栈。

实际上还同时有栈溢出 UB 这种无解的设定。

3 一些本不适合内建的特性相关问题

内建特性在兼容性和扩展性上尤其容易出现问题，特别是存在不够显然的实现的时候。

3.1 异常不得不被实现为内建的，某种意义上是缺乏上述两者的直接结果——这导致了另外的问题：

- 例如ABI兼容性——同一个体系结构和操作系统的同一版本的同一个实现，使用不同不同异常模型时仍然可能不兼容
- 再如 WG21/N4049

3.2 面向对象支持——关键是面向对象本身就没有个相对统一的认识。

- 内建导致之后扩充多分派等需要顾及的太多而最终放弃，远不如 CLOS 之类的库解决方案灵活
- 同样在一些方面导致和加重了ABI问题

4 阶段(phase) 相关

某种意义上是缺乏同像性的副作用。

ISO C++ 规定的实现阶段数量过多，且过于琐碎。

这导致细节难以处理，也难以提高实现质量。

4.1 预处理被标准化，但是细节比较混乱，导致了很多演进上的问题。

- 像 `#ifdef` 和 `#if defined` 的冗余

4.2 缺少可用的模块系统

ISO C++17 似乎不能保证引入……

4.3 存在语言链接支持却缺乏可移植的语言嵌入。

只有一个 `asm` 还不指望能用。

5 至于文法/语法嘛……

本来懒得说，不过因为逗到一定境界也顺便提一下好了。

5.1 兼容 C 的渣声明符中缀语法

特别地， C++11 引入 `trailing-return-type` 后也没法放弃旧的语法，并存的结果是增加复杂性。

5.2 比 C 更多的文法歧义

最著名的坑应该就是函数声明优先于初始化对象了。另外还有几个类似的地方。

5.3 标点问题

因为基本源字符集的限制，使用大于(`<`) 和小于(`>`) 的尖括号(angle brackets) 代替了数学上习惯的看起来更扁一点的括号(chevrons) “〈”和“〉”。这导致了极大的解析上的复杂性，同时还需要增设特例规则（记号 `>>` 和记号序列 `> >` 歧义）。

5.4 少数 C 的语法的不规则延伸

如 `new` 表达式。

除了单独的消歧义规则的复杂性外，特异的语法规则导致用宏替换实现一些操作更困难，也导致语言规范的复杂（有多少人能分清 type-id 和 new-type-id 呢）。

[科普]为什么指针是个糟糕的语言特性

Created @ 2015-08-23 07:27, rev2 2015-08-23 14:37, markdown @ 2015-09-16.

0 引言

本文主要是写给自信知道“指针”是什么玩意儿的读者看的。最好看完再评论。（我大致上能确定 99% 以上的并没有足够重视这里的坑，不够了解现实。）

不知道指针是啥的虽然看了基本是浪费时间，不过至少务必记住：不是所有叫“指针”的东西都是一回事。

1 什么是指针

本文所谓的指针(pointer)，是指C和C++等语言中的内建的语言特性。

在不同范畴中指针这个概念有所不同。在体系结构规范中，指针指称特定的整数字节地址或者两个地址的差（地址偏移量），是整数数值；而在C和C++中，作为核心语言特性支持的指针是一类类型的统称。这两种完全不同的概念经常被混淆，造成一些稀里糊涂的问题（和数组混在一起的时候尤甚）。除非另行说明，本文总是指后者，并不对此进一步展开论述。

C和C++中，指针（右）值是具有指针类型的（右）值。指针值有时也会被和指针混淆，但在健全的理解下通常能消歧义，因此问题不大（数组也有类似的情况，但涉及转换，问题相对严重）。为清晰起见，在这里不会不加区分地使用。

注意，C++的成员指针(pointer-to-member) 明确地不是指针。尽管它的数值表示在一些情况下可能被实现为地址的偏移量，但语言中并不存在这种保证，实际也通常不那么简单。重复：成员指针不是这里讨论的指针。

此外，C++中，除了作为语言特性支持的内建(builtin) 指针，也有所谓的智能指针(smart pointer) 。后者在概念上也被 ISO C++11 以来的标准库正式支持。这里讨论的指针不包括这些智能指针，尽管后者和主题相关，并且会在下文重点讨论。

2 什么是设计

这里讨论的设计，是指语言的设计，也就是语言规则的作者决定语言特性中应该存在什么和不存在什么的决策之下的抽象结果。

用户如何使用指针即语用问题是和本文主题相关的问题，会一并讨论，但和这里的设计是两个不同的话题。

3 什么是糟糕

糟糕是一个形容词。

形容设计的糟糕从两个递进的视点得出：对照设计要解决的问题，即需求；对照同类解决方案，即语言中的其它特性或应用领域有交集的其它程序设计语言中的特性。

通俗地，糟糕以“不好用”和“并非不得不好用”来表现。

注意因为语言规则之间的相互作用，是否“好用”或者说要解决的问题，须结合使用场景下的其它问题一并讨论：一项特性即便能很好地解决某些问题，但若几乎总是引起其它难以回避的问题，那么至少是“不够好用”的；而要造成的问题麻烦到一定程度时就显然“不好用”了。

4 指针有什么用

在说明不好用之前，首先需要了解有什么用。

这是一个发散的语用问题，但大多数用法都很浅显，清楚语言规则就并不难归纳。

4.1 指针和地址

C/C++ 的指针值和体系结构中的所说的指针（地址或地址偏移量）的基本作用类似，它用来指示数据存储的位置。

以体系结构的接口实现 C/C++，可以轻易保证相同类型的指针值到地址的映射是单射，即相同指针类型的指针值的不同的数值表示可以总是找到不同的地址对应，这样就可以在整数算术和关系操作的基础上毫无额外代价地定义指针算术和关系操作；而指针上的操作符 `*` 抽象的正是间接寻址操作。这就是一些用户口中的所谓“接近底层”。这种简单直接实现的最大好处就是容易以非常小的代价生成针对特定体系结构的代码。

一个需要注意的关键不同点在于，C/C++ 作为强类型语言（这里的用法也比较乱，指的是原本意义——默认具有静态类型检查），其中的值(value) 脱离类型讨论并没有意义，指针值也不例外。对象指针可以进行算术操作，但和整数地址算术的含义并不相同，这受到具体指针类型的影响——例如，`T*` 和整数的 `+` 操作和 `sizeof(T)` 相关；而函数指针并没有类似的意义。此外，需要不同间接操作层次的值如 `T*` 和 `T**` 也可被明确地静态地区分，光靠地址并不能做到这点。

然而，因为体系结构（硬件）实现的惯例，这个差异在往往能被利用（典型地，基址变址寻址），生成相对高效的代码。这是语言中保留指针算术的用途之一。另一方面，把地址相关的整数数值明确和一般的整数值区分，也明确了目的，使静态检查非预期的混用成为可能，有限地提供了类型安全性（例如，指针和指针不能相加）。

通过两个地址，或一个地址和表示字节大小的一个非负整数就可以标识出地址空间的区间范围。把地址替换为对象指针、字节大小替换为长度（指针值指向的连续元素的个数）同时限制取得指针的手段，能保留这种标记连续存储区域的功效，同时提供一定的可移植性。这种连续的存储在类型系统上被抽象为数组。不过应当注意，在可移植的要求下，实际上指针的语义依赖于数组。完全绕过数组的存在任意地构造一个指针值不能保证指向有效的对象或函数，进行间接操作基本上总会导致未定义行为。

另一个关键不同是空指针值(null pointer value) 并不保证有特定的地址对应，见下文。

4.2 存储资源管理

因为一个对象指针和长度可以用于表示连续的内存，而对象（排除 VLA ）的大小能在翻译时静态确定，所以在已知大小的存储区域可以用一个对象指针值直接表示。

ISO C 标准库的 `malloc` 和 `calloc` 以及 ISO C++ 标准库的 `::operator new` 和 `::operator new[]` 等的返回值是典型的实例。

这里大小是由存储分配另外保存，这样释放时仍然只需要传递一个指针值即可。ISO C++14提供了 sized deallocation ，不过这并非强制。

4.3 参数传递

因为从分配函数中取得的指针表示的存储并不会如自动变量一样会被自动回收，同时指针有间接操作，而指针值作为对象类型的值可以作为参数传递，因此传递指向对象的指针值配合间接操作就可以模拟传递对象的引用。

4.4 基于存储的迭代

因为对象指针能表示存储位置，连续存储的布局由存储模型（以及基于数组的语义）规则限定，适当类型的指针值进行算术操作可以双向顺序迭代乃至随机访问连续存储的对象。

4.5 空指针值

指针类型是可空类型(nullable) 类型，约定特殊的空指针值表示不指向任何对象或函数，但可以进行有限的比较。

可空类型很容易用来表示可选(optional) 值 ：约定空指针表示值不存在，非空指针指向的对象或函数即存在的可选值。

空指针值还可以表示哨位(sentinal) 即迭代终止的标识。相对于具体存储区间结束的指针相比，空指针值是通用的，并不需要根据特定的区间使用不同的值。

注意空指针值的存储表示不一定是整数零值（这再次体现了人为预选的数值和地址的无关性），尽管使用零值一般能有更好的初始化性能。

5 指针为什么不好用

既然标题已经确定了指针设计的糟糕，那么在“不好用”上自然有充分的理由。

总结就是，指针看上去能干很多事，但没一样事是完全干好的，还有的事（比如声明语法）甚至在一般意义上就特别差。

讽刺的是，第一个大规模使用这种指针的 C 语言作为UNIX系统的实现却完全违反了 UNIX 程序鼓吹的模块化设计哲学：只做一件事，并且把事做好。

为什么“程序”应当遵守的原则，分解到实现语言的特性的层次上，就可以罔顾设计原则乃至表现得相反了呢？难道这里不更应该体现接口的可组合性吗？耐人寻味。

5.1 使用的意图

首要的原罪就是指针能干太多事了，导致如果只需要其中的某些功能子集（几乎所有情况都是这样，实际上也不可能全用上，见下文）就不容易看清楚代码在做什么，也就是任何“正常”的使用与使用其它替代实现手段相比，都很容易明显损害代码的可读性。

要避免这点，要么放弃使用指针而使用其它替代，要么就不得不以文档（包括注释）等形式来把这些接口规格中大多不必出现的琐碎细节约定清楚。后者很容易显著增大实现和维护的工作量。

5.2 易错

因为意图不明的关系，使用指针的代码比使用其它更清晰的替代的代码更有机会错误，而指针本身的静态类型检查对此爱莫能助。

最显著和严重的错误可能是对于存储资源管理的错误。

注意C/C++语言要求去配函数的指针值参数若非空，则必须和适当的分配函数的返回（指针）值相等且不能以相同的值调用去配函数超过一次，否则程序行为未定义。

因为指针值并不保证翻译时确定，静态检查对此类误用效果很有限，要想安全使用且不泄露资源，用户必须清楚使用的指针是否可以被释放，然后准确保证从分配函数得到的非空指针值恰好作为参数调用正确的对应的去配函数一次——这里是否可以释放的所有权(ownership) 信息并没有编码在指针的类型之中。

注意，单看一个指针值，有或者没有所有权是确定的，不存在第三种状况。鉴于这两种状况互斥，因此一个指针值不可能同时是表示存在所有权的资源指针和表示不存在所有权的资源视图/观察者指针。这也就是上文说“不可能全用上”的原因。

然而事实是明确持有一个有所有权的指针，需要释放时，光看指针根本没法知道该使用哪个去配函数……更有甚者，其实光从指针上根本就看不出有没有所有权。

如果一个返回指针值的函数不幸没有文档描述清楚用户该如何处理资源释放问题，就面临了两难的风险：调用错误的去配函数或重复释放导致未定义行为，或者放置不管而至少产生泄漏的后果。

可能就是因为这样， WG14 （ C 标准委员会）有一条不成文的规矩：返回指针的函数总是不带所有权——也就是用户不应该释放这里的资源。

然而就连 Austin Group （起草 POSIX 标准的作者）对此都并不买账（更别提 GNU 等了），造成了接口设计上的冲突（详见 WG14/N1174 ），可见这条默许的规则在 C 用户的范畴内整体上行不通。

用户该何去何从？看脸……（没有接口文档的自己踩坑怎么死？看着办。）

5.3 不必要的负担

这里最明显的例子是明明静态确定在不需要空指针的情况下不得不判断指针值是否为空，给程序运行带来不必要的开销。

所谓的“空指针”滥觞于 C.A.R.Hoare 在 1960 年代的 ALGOL W 语言的发明。2009年，Hoare 在一个会议上为此道歉，原因是空指针特性引发了很多程序设计中的错误和漏洞。

盲目省略空指针值检查的导致使用指针间接操作的值引起未定行为的错误威力并不比上面资源管理的错误来得小。因此一旦接口沾染了指针，事情就复杂了——最容易的修复就是放弃使用指针这样的可空类型。

5.4 语法噪音

上面说的都是语义直接相关的语用困难。

事实上，即便不考虑语义问题，经验表明光是 C/C++ 的指针语法（严格来说不光是指针自己的问题，还有数组、 C++ 的引用和 C++/CLI 的句柄等，都属于此类）也相当反直觉了。大部分用户遇到嵌套的指针声明符甚至都不能一下子看明白边界，更别说表示什么意思了。

对这个问题的主要变通是使用 `typedef` 。但未必每个接口都会老实用——比如 ISO C 的 `signal` 函数就没有用。所以遇到了用户还是要硬着头皮看。另外还可能有同时有使用 `typedef` 和不使用 `typedef` 名称并存然而两者等价的局面，此时用户就得当人肉编译器自行验证 `typedef` 和复杂声明符的等价性了……

而现代的编译器也没能利用这样的语法带来简化。

鉴于这种看起来精巧实则无用的设计带来的困难，Bjarne Stroustrup 等在 C++ 尝试引入更直白的语法。但是，虽然 `trailing-return-type` syntax 是引入到 ISO C++11 里了，兼容 C 却不能排除旧的语法，结果就是对用户来说存在两套不完全兼容语法要学，编译器也得把两套语法都实现这样一个混乱局面……

5.5 语义噪音

同样因为意图不明的关系，要让不同用法之间存在差异变得困难了。

举例来说， C++ 不需要内建指针模拟对象引用传递参数，所以看到 `->` 和一元操作 `*` （重载另说，但不抽风的重载不应该和这里的清晰性背道而驰）就可以大致确定此处进行的是非平凡（模拟参数传递）的操作。

考虑到模拟引用参数也必然不需要空指针值，这样一来差距更明显。

5.6 抽象的无能

或许抽象能力的缺失才是最大的现实问题，因为关乎高级语言的本质目的，而并非特定的个别需求。

一个例子是，迭代存储连续的序列用算术操作，为什么同样是迭代，链表就不能类似的语法呢？

不过只是“不好用”的角度并不容易集中体现这一点，此处先略过。

5.7 互操作性

和体系结构的交互或许是指针唯一合适的领域了。不过，这依赖于实现的假设，因此操作起来并不那么有普适性。

即便平时鼓吹“硬件友好”“接近底层”，事实上 C 就不存在对地址空间的抽象，还得靠厂商或者 WG14/N1169 这类几乎名不见经传的扩展。

倒是 C++ 标准库的分配器机制本来有要支持上面扩展的考虑不同的指针，虽然后来都流行平坦地址空间然后这个需求就死得差不多了……

一个根本硬伤是，相同类型指针值到地址的映射是单射而不是满射——也就是任意一个地址即便在体系结构和实际机器的环境下允许，也有重重限制，根本不保证能用能自由操作的指针表示。

这样，关键时刻到底还得上体系结构相关的扩展乃至汇编和/或机器语言……（什么硬件友好接近底层，见鬼去吧(╯`□′)╯︵┻━┻！）

5.8 理解的混乱

事实证明，指针自身的微妙规则以及和数组之间看似说不清道不明的关系给教材编写者以及初学者带来了极大麻烦。

不管是 Bjarne Stroustrup 鼓吹的 teachability 还是一般用户期待的“易用性”，指针的语法和语义规则都是重灾区。

总体来看，这种的问题的根源来源于指针这项语言特性自身的设计——包括是不是真的适合作为核心语言特性这点。

5.9 谁来承担责任

可笑的是，缺陷这样明显的语言特性，一边在被各种集中地滥用和误用，一边被井底之蛙吹嘘为“ C 的灵魂”骗更多不知情者上当……

容忍这样的缺陷和制造混乱代码的作者通常是同一拨用户。对于不良语用导致的后果却往往由合作的理解更透彻的维护者承担，把本可以满足更多现实需求的时间花在给脑残粉的烂代码擦屁股的破事上。

这是有多不公平呢？

6 “指针”必须这样不好用吗/不用指针用啥

如果不限于内建指针，答案是否定的。

从指针几个有用和常用的使用惯例来看，搞清楚真实需求之后，很容易设计出更安全好用的机制。当然，得有足够的其它核心语言特性支持，类型系统羸弱的 C 只能靠边站。

对这里的缺陷修正得比较彻底而又比较流行的例子主要就是 C++ ，同时 C++ 也保留了指针的操作，反而更有必要澄清什么时候不适合用指针。所以以下以 C++ 为例（涉及的主要特性，其它现代语言，即便没有指针也大多有对应）。

6.1 了解意图、避免常见错误和提升可读性

若需要间接操作表示资源，使用带所有权的智能指针。同时可以自动管理资源，避免资源泄漏。不加封装地使用内建指针意味着更混乱的代码路径，通常是糟糕的代码。

若需要间接操作表示不带所有权的资源视图，使用不带所有权的特定指针类型，如 WG21/N4282 提议的 `observer_ptr` 来帮助表明意图。（这里使用内建指针的问题相对比较小，在没有其它选择的偷懒情况下，使用内建指针相对来说能够被容忍，因为带有所有权的指针已经被其它智能指针区分出去了。）

若需要传递引用，直接使用内建引用。在需要复制引用的场合，使用 `std::ref` 之类的包装。内建指针在此本质上毫无必要，并且无法使用大部分其它设施（只有 `std::bind` 等一些少数例外）。

若需要可空类型，使用 WG21/N4480 等规范的 `optional` 类型。（内建指针仍然是个可以忍耐的替代，但并不推荐。）

若需要迭代操作，使用迭代器(iterator)。迭代器同时有更好的类型安全性、适应性和可扩展性。指针作为随机访问迭代器的特例是可以被使用的，但仍然应当小心行事。

通过划分典型应用场景，就基本解决了上面的最麻烦的一些问题。除了静态区分存在和不存在所有权相互矛盾之外，以上类型也是可以组合的，因此同时需要多种意图也不需要使用内建指针。

6.2 抽象能力和可扩展性

这集中体现在智能指针和迭代器与内建指针的对比之上。

内建指针的语义基本是被核心语言规则写死的，它并不能实现智能指针这样用户自定义资源所有权管理策略，以及迭代器这样的适配于不同实现构造的序列上。因为过于特殊，可以说是相当地无能。高下立判。

通过迭代器类别(iterator category)的抽象层次和 tag dispatching 这样基于重载（说穿了，一种模式匹配）的技巧，还能实现对不同性质的序列静态自动选取最优算法。不知比指针高了哪里去了。

当然，内建指针和典型体系结构实现之间的能力仍旧没有被取代。但指针在真正底层（比如说，地址空间）的抽象仍然一直是个坑。而且这明显不是高级语言的本职工作。如果不是照顾兼容性，让厂商实现成扩展并用标准库包装，说不定还不会像现在那么混乱。

6.3 约束更强的设计

（现代） C++ 是强烈强调静态类型存在感的语言。这种设计有利有弊，但从实践效果来看，正确地使用能够发挥静态类型检查的优势，是当代软件工程实践的重要趋势之一。（静态类型当然有非常鸡肋的地方然而现实是大部分用户根本连边都碰不到……注意缺乏元数据是 C++ 和标准化的锅，不是静态类型的锅。）

但是 C++ 限于历史包袱（兼容 C 、兼容现在各种代码），即便比 ISO C 敢于甩手扔包袱，也得考虑一下现实影响。在这个意义上，用户相对较少的小众语言以及新设计的语言就没有那么多顾虑，能将有目的的设计刻意发挥得更充分。

举两个稍微不怎么小众的例子。

一个是 Haskell 。应该说重点不纯粹是静态类型的问题，而是在类型系统的设计上使用了对静态分析友好的较为系统化的设计。（而并不是像 C++ 那样一小坨一小坨地加特性，而这里最大坨的 Concept 被否了……）

当然这货主要用于开眼和拓展想象力，因为默认求值策略过于标新立异实际上不适合通用的需求，在 DSL 以上的实用还是算了。

另一个是 Rust 。嗯，设计的一个主要目标是取代 C/C++ ，应该还算是比较现实（？）的。在这里值得一提的是有不少设计把上面的策略整合到核心语言特性上去了并且有系统的理论支撑，比如 linear typing 是对 C++ 的 `std::unique_ptr` 强化 。

姑且不论大杂烩的实用程度，这在科普上比较有意义。

6.4 复杂性谁来买单

有的用户可能会说，这么复杂，还是用内建指针直接偷懒算了。

对此我只能表示呵呵。你真有自觉到完全写清楚各个层次的接口文档表明语用？——注意，各个层次，包括现在当成实现细节而将来可能被接手的其他维护者当成内部接口使用的任意层次的“接口”。

如果：

(1)因为非自身原因只能用 C 这等无能玩意儿的而且真做得到及说服了其它倒腾这坨代码的（如果有）也同样做到上面所说的自觉，或者——

(2)保证这坨代码不流入公众视野充实反面教材，同时实现者保证必要时时刻忏悔生产垃圾多出来的碳排放

那么当我没说。

否则……思想有多远就给我滚多远。

又不是叫你发明新语言特性自己实现编译器，都敢倒腾“底层”语言了，了解基本需求和解决方案这么点简单的份内之事都做不好还有脸生产垃圾污染环境让人擦屁股来添乱？

还是有谁逼你用这坨容易炸的东西了？（不懂适应现实？那么饿死活该。）

注意，业界从来不缺猪队友，少一头的确照样转（蠢代码照样蠢）。

7 结语

略。

语用和科普

[科普][FAQ]MinGW vs MinGW-W64及其它

Created @ 2014-07-25, r2 rev 2015-09-15, markdown @ 2015-09-15.

部分参照备忘录原文。

试试问答体。首先得绕个远路，从 Win32 开始说起，否则之后容易乱……

什么是 Win32 ？

嘛，32 自然是指 32 位了？不一定。

正式地说， Win32 主要是指跑在 Windows NT 内核上的 Win32 子系统。现在 x64 的 Windows 上的大部分程序也是跑在这个子系统上的， `system32` 目录也没叫成 system64 。

尽管 32 的语源的确来自于“32 位”。

那么为什么还有 Win64 ？

这倒可以肯定，这里的 64 是指 64 位目标平台，因为没有上面的那种歧义。

有一点值得注意，在 MSVC 中， 32 位环境（当然是说跑的 Intel 兼容CPU的PC）预定义宏 `_WIN32` ，但 64 位环境同时预定义了 `_WIN32` 和 `_WIN64` 。

顺便，通常 64 位主要指 x86_64 （微软称为 AMD64 ，这个兼容 x86 的基础架构一开始的确是AMD先搞出来的，后来才有 Intel EM64T ）。

64 位 Itanium 也有 `_WIN64` ，不过一般见不到且跟MinGW没什么关系且现在都不正式支持了，不管了……

对于 MinGW 来说，这里也有类似的坑：预定义宏得先优先检查 64 位的。实际情况更加复杂，另说。

MinGW 和 MinGW-W64 有什么区别？

这是个关键问题，但是……是个很长的故事。没有铺垫不好回答。

首先， MinGW 是 GNU 工具（包括编译器 GCC 和 GNU binutils 和调试器 GDB 等）在 Win32 上的一个移植，是从 Cygwin 里 fork 出来的。当初只考虑 32 位。和 Cygwin 相比，不强调 POSIX 兼容性而相对强调性能和减小依赖。

具体来说 MinGW 除了以上工具外，还提供了一个适配于 Win32 的运行时环境。其中 C 标准库实现的二进制文件直接用微软随 Windows 分发的 MSVCRT 。 MinGW 自己的运行时库依赖于MSVCRT和其它系统库。

而 MinGW GCC 依赖于 MinGW 运行时以及 libgcc 和其它系统库。编译出来的程序一般也要依赖这些库，所以才会写死在默认 specs 里（可以用 `gcc -dumpspecs` 查看）免得用户随便编译链接个程序还得手动指定一大堆 `-l` 选项。

用三元组表示目标平台，当年的 MinGW 是指 i386-pc-mingw32 。这里 i386 也可以是 i486 等等……总之是 32 位 x86 指令集架构的名称；中间的 pc 可选，表示厂商名； mingw32 表示系统名。特别注意，事实上成为标准的“专有名词” mingw32 里的 32 是固定的；另外，所有这些大小写一般也是固定的。 GCC 等的源码配置里面也有硬编码进去。

然后，因为只支持 32 位，有人觉得不够用。这里的一个主要人物，就是现在 MinGW-W64 的主要维护者 Kai Tietz 。因为工作需要重新实现提供 x64 支持的 MinGW 后提交到上游但被拒绝，于是 fork 为单独的项目，这就是 MinGW-W64 的由来。

可见， MinGW-W64 和原版 MinGW 有所渊源，但是独立的两个项目。

W64 虽然用意是Win64，但是也算是个专有名词，在三元组里占据厂商名，例如常见的：i686-w64-mingw32。（在GCC源码的配置文件中， *-w64-mingw32 和 *-pc-mingw32 是分别对待的。）

MinGW-W64 是同时支持 32 位和 64 位的，甚至还支持 32 位和 64 位的交叉编译（启用 multilib 支持的 MinGW 发行版例如 mingw-builds 可以用 `-m32` 或 `-m64` 指定）。

显然， W64 和支持的架构无关。上面 i686 就不是 64 位的平台（而且可以看出这里的 32 也和架构没关系）。支持 64 为的对应三元组是x86_64-w64-mingw32。（另外 w32 是 GNU 惯用的对 Win32 的略称，也沿用到包括 MinGW 在内的一些项目中——如 w32api ，可能造成一些额外的混乱。）

……容易让人头疼的是，这两个项目现在都没死，偏偏还很容易因为这些字面上的原因搞错。为了下文描述方便，原版 MinGW 称为 MinGW.org 。

这里有一点非常重要：只有 MinGW-W64 是 GCC 官方支持的（尽管 mingw32 平台是二等公民）。 Kai Tietz 拥有 GCC 官方 repo 的提交权限。

所以，使用 MinGW-W64 的 GCC 一般比 MinGW.org 有更新更全面的支持，所以现在一般推荐 MinGW-W64 发行版。

说到这里……维护 mingw.org 的 Keith Marshall 还和 Kai Tietz 等GCC官方人员在 bugzilla 上对喷过 。其中 Keith Marshall 对 MinGW-W64 使用 MinGW 一名造成混淆表示愤慨。嘛，这倒也是事实。

当然，也不是说 MinGW.org 就一无是处了。 *-w64-mingw32 原则上向后兼容 *-pc-mingw32 ，不过也有一些接口上的差别。 BSD 流的 `DT_*` 在 MinGW.org 上能用，在 MinGW-W64 的就没有。（虽然 `DT_*` 也不怎么推荐用就是了……）

什么是 **MinGW** 发行版(**distribution**) ？

这个说法习惯上可以说是从 Linux 等软件中借用过来的。

类似 Linux 内核，不管 MinGW.org 还是 MinGW-W64 ，本身都是相对集中于特定软件包（ MinGW 运行时库）开发的项目，并不着力于提供整个开箱即用的环境。

因此除了官方的一些编译版本外，有很多人基于 MinGW 运行时上进行定制封装供用户下载整个环境，有的还提供包管理服务等。这就是发行版。一般提供直接解压加上 `PATH` 就能用的环境和/或安装包。

早期比较著名的有 TDM-GCC 、 rubenvb 等。以前用的 MinGW.org ，不过现在主要转到 MinGW-W64 上来了。

比较新的发行版，一开始就着眼于 MinGW-W64 。最著名的发行版之一应该是 mingw-builds ，基本上近年来（ GCC4.7 以后） Windows 上能用支持最新版本最快的，支持交叉编译。

mingw-builds 一开始在 sf.net 上有自己的项目，不过后来表示要求加入 MinGW-W64项目 作为 official builds ，所以停更了，更新都在 MinGW-W64 里面，不过残念的是好像到现在 MinGW-W64 看来都不提供唯一的官方发行版，所以还是叫做 personal builds 。

另外提一下还有微软 VC++ 标准库（ Dinkumware 生产）维护者之一 Mr.STL(Stephan T. Lavavej) 个人的发行版，很早就在默认 specs 里加了 `-std=c++11` ， 而GCC 5 改用 `-std=c++14` 。（官方 GCC 6 会用 `-std=gnu++14` 。）

还有 MSYS2 项目的 MinGW 发行版（这里可能有新的混乱，下文再说）， 也是 mingw-builds 一伙人搞的， 4.9.1 比 mingw-builds 更新还快几个小时。

其它发行版可以参照 mingw-w64.sourceforge.net ， 更新相对没那么快。

最后，不嫌闲着蛋疼也可以自己编译。不过迫不得已外最好别这样做（ GCC 的编译过程和 hacking 实在无力吐槽）。重复一遍，非常不推荐。

上面为什么要强调更新呢？

如果不想使用新的特性生成更高质量的代码，其实也没必要盯着上面这么多版本混乱的 MinGW 了。即便要兼容性，也可以用古董嘛（逃……

对于 C++ 前端来说 MinGW 尤为重要，现阶段根本没有能顶替的。作为系统默认 ABI 新锐代表的 MSVC2015 ——前端还是残的……各种 bug 。

GCC 也有各种傻缺 bug ，不过至少在前端来说，程度上绝逼打不过 `cl` （ Microsoft C&C++ Optimizing Compiler ）。

VC++ 调试支持当然好得多，但是编译器一坑爹集成调试再好也没用了。

嘛， Clang++ ？ libc++ 什么时候能在 Windows 上跑顺再说——即便这样 MinGW 兼容的还是得依赖 MinGW 的 libgcc 。至于和 VC++ 兼容的 `clang-cl` ，看起来还在折腾微软的坑爹ABI黑箱（这没像大部分平台上 GCC 用的 Itanium ABI 公开文档），一年半载别指望了。

什么是异常模型和线程模型，用哪种比较好？

这两个都是对于 C++ 实现（ G++ 、 libgcc 、 libsup++ 等等）而言的。

首先，异常模型。 C++ 标准没规定异常怎么实现。 MinGW GCC 使用的 Itanium ABI 也没规定必须怎么实现（但规定了一些公共接口），这部分由实现自行考虑。

GCC 一般提供了 SjLj （ C 的 `setjmp` / `longjmp` ）实现的 stub 。对于 x86 ，允许使用 Dwarf2 调试信息的实现。两者的区别在于 sjlj 比较通用，但是即便不抛出捕获异常而只是使用异常中立的风格隐式传递异常，也有运行时开销。而 Dwarf2 兼容性（考虑多层 C++ 和 C 的 DLL 互相调用来看）相对较差，但没有这种开销。

两者 ABI 并不兼容（知道 C++ 坑爹了吧，不仅不同实现不兼容，同一个编译器同一个平台自己都能跟自己不兼容……）——前者依赖类似 `libgcc_s_sjlj.dll` 这样的 DLL ，后者则是类似 `libgcc_s_dw2.dll` 这样的。旧一点的可能也没有这种后缀差异。

另外， libstdc++ 作为 C++ 标准库实现显然依赖异常，但名字一样的 DLL 可能依赖的不是同一种。所以混用多个版本 MinGW GCC 且没把 `PATH` 清理干净的时候容易出现找不到符号定义导致链接失败。这还不是最坑的，有时候 `gdb` 载入不同位置的 DLL 在运行时挂掉，还不只是一个 `PATH` 的问题……这种情况下先拿 Sysinternals 的 Process Exporler 之类的工具看看进程加载的 DLL 是不是预期的再说。

为什么说要有这么坑爹不兼容的，像 VC++ 一样用一种不就好了……其实 Win32 x86 上最理想的应该是和 VC++ 一样基于 SEH （ Windows 结构化异常处理）的实现，但是 Borland 关于这个的专利才没过期几天……所以你懂的。

x64 上没专利的麻烦，有 SjLj 和 SEH 的实现，一般还是 SEH 。

第二，线程模型。

主要有两个， Win32 和 POSIX ，对标准库线程的支持不一样 。

Windows 线程 API 和 POSIX(pthread) 有很大不同，而 ISO C++ 的 `std::thread` 为代表的接口是很接近 pthread 的。

所以在 libstdc++ 上实现这些接口，首先依赖的是 pthread 在 Win32 上的移植 libwinpthread ，也就是使用 POSIX 线程模型。因此发布的时候需要带上 `libwinpthread-1.dll` 之类的dll。

至于 Win32 线程模型， GCC mailing list 是有提过，不过到现在还是没实现。也就是说 ISO C++ 的实现是残的，没法用。如果只打算用 Win32 多线程 API 倒是的可以用。

所以取决于具体需要。要兼容性好点的一般还是 POSIX 。

另外还有单线程的 single 模型…… Windows 上应该没啥必要用。

什么是MSYS，和MinGW有什么区别？

MSYS(minimal system) 原本是 MinGW.org 项目的一个组件，旨在 Windows 上提供一套类 UNIX shell 为基础的“系统”。它本身不提供编译器或者大小写敏感的文件系统支持（其实 NTFS 倒是支持这里的“ POSIX 语义”，但基本没看见有谁用……）。

和作为原生 Win32 程序的 MinGW 不同， MSYS 环境下编译的本机程序依赖于额外的特定的 MSYS 运行时，更接近 Cygwin （强调 POSIX 兼容性），会有性能损失（但一般意义上比 Cygwin 轻量）。对应的三元组是 *-pc-msys （通常其中的 pc 可以省略即缩写为 *-msys ）。 MSYS 提供了一个 sysroot 环境（下面有 `/bin` 和 `/etc` 等），因此移植 POSIX 环境的程序一般更方便。

所以常规的实践是，如果只是开发 Windows 程序，能用 MinGW 就不要用 MSYS 原生的编译器来构建。当然，使用 MSYS 上的 `sh` 等工具还是没问题，跟 GNU 工具配套怎么说比 `cmd` 好用。（虽然也有不少琐碎的兼容性问题。）

什么是MSYS2，MSYS2上的MinGW发行版是怎么回事？

字面意思，MSYS 2.0 。比起 1.0 来说更加像 Cygwin （例如 `/etc/fstab` 配置）。项目在 sf.net 上托管。

其中的一个特色是基础系统附带 ArchLinux 移植的包管理器pacman，可以同时独立部署 `/mingw32` (i686-w64-mingw32) 和 `/mingw64` (x86_64-w64-mingw32) 下的开发和运行环境。注意和 mingw64 并列时 mingw32 自然指的不只是三元组的最后一项了。

下载依赖相当方便（就是没有靠谱的镜像时网速可能非常拙计）。具体使用参考 Arch Linux Wiki。对应的交叉编译环境在 Arch Linux 上也有官方的包支持。

虽然 MSYS2 提供的 MinGW 上主要的编译器不直接支持交叉编译，不过可以同时装不同的目标平台的编译器（现在也有交叉编译器的包，没测试）所以不是什么问题，一定程度上比 mingw-builds 的 `-m32` 和 `-m64` 来说更加稳定靠谱。（现在也另外提供支持交叉编译的包。）

只提供 Dwarf2 异常模型和 POSIX 线程模型对于成套系统也不是什么大问题。包虽然比不上 ArchLinux 那么丰富不过常用的很多都有，免去自己编译的麻烦。打算长期使用 MinGW 和相关工具的，推荐使用。 虽然滚挂了也没多大事，不过版本是个问题。如果需要特定版本的 GCC 就不适用（比如规避GCC 4.9的坑爹bug……），除非有耐心自己找到 `.xz` 手动安装。

相关配置（包括 `pacman` 的一些中国大陆镜像）可以看这里 。

部署程序需要提供哪些文件？

Windows 默认安装自然不带 MinGW 运行时环境，所以除了编译出来的程序和可能附带的数据，一些dll是要准备好的——除非有耐心折腾全部静态链接。

不同版本不同语言不同编译器编译出来的东西都不太一样。最简单暴力也可靠（？）的方法就是复制可执行程序到没装环境的白板测试机上看少了哪些东西（不过未必一目了然）。

简单可靠的方式是用 Dependency Walker 等工具查看依赖。

对于 C++ 程序，除非不用 POSIX thread 可以省掉 libwinpthread ，一般至少得确保上面异常模型和线程模型讨论中提到的三个 DLL （注意就算不显式使用标准库，编译器生成的代码也可能用到——典型的如默认 `::operator new` ，所以得带上 libstdc++ ）。

现在还有什么新坑？

就提一个GCC 4.9的问题。

GCC 4.9的 LTO （链接时优化）默认使用新的目标文件格式，生成的文件不包含冗余的二进制代码。 但是 LTO 有特定的 phase （内部会调用 `make` 多编译几个 pass ），传统的静态链接器(`ar`) 不知道这里的约定，所以原来好好的东西，升级 4.9 以后开了 `-flto` 就可能找不到符号链接失败。

许多 MinGW 发行版仍然没实现 `gcc-ar` （运行会提示没支持 linker plugin ）。兼容旧版本的行为还得加上 `-ffat-lto-objects` 编译选项。